МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ М. В. ЛОМОНОСОВА
ФИЛИАЛ МОСКОВСКОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА
ИМЕНИ М. В. ЛОМОНОСОВА В ГОРОДЕ САРОВЕ

Параллельные методы решения задач

Лабораторная работа:

«Параллельная реализация решателя СЛАУ для кластерных систем с распределенной памятью»

Выполнил студент
1-го курса магистратуры
Козлов Н.М.
Группа: ВМ-124
Дата подачи: 28.05.2025
Вариант: А2

Саров 2024

2.1. Описание задачи

Цель данного задания: реализация решения СЛАУ, хранящегося в CSR формате с использованием технологии MPI.
Поэтому нужно:

1) осознать масштаб проблемы;
2) ~~закрыть лабу и спать спокойно~~ создать массивы L2G, Part, G2L для установления взаимосвязи между кусочками сетки;
3) адаптировать Generate, Fill, Solve под особенности MPI;
4) написать схему обменов Com и воспользоваться ею в Update перед SpMV;
5) сформировать отчёты о работе программы, сходные предыдущим, а также отчёт о контрольных величинах, которые сравнить с OpenMP версией.

1.2 Описание программной реализации

Для решения задания были добавлены следующие функции:

```
//Получение глобального индекса по локальному внутреннему
int Global(const int index, const int Nx, const int Ny, const int px, const int py, const int Nx_loc, const int Ny_loc)
//Массив взаимосвязи всех внутренних-интерфейсных-гало элементов с их глобальными индексами
void Local2Global(const int Nx, const int Ny, const int k1, const int k2, const int px, const int py, const int Nx_loc, const int Ny_loc, int& N, int& No, int*& L2G, int*& Part)
//Обратный для Local2Global массив, чтобы узнать локальные позиции по глобальным индексам
void Global2Local(const int* L2G, const int N, int& Np, const int Nx, const int Ny, int*& G2L)
//Схема обмена граничных элементов для обновления гало
void Comm(const int N, const int No, const int* IA, const int* JA, const int* Part, const int* L2G, const int* G2L, const int px, const int py, const int k1, const int k2, const int Nx, const int Nx_loc, std::vector<int>& Neighbours, std::vector<int>& NeighboursSend, int*& RecvOffset, int*& SendOffset, int*& Recv, int*& Send, std::vector<int>& Interface)
void Update(double*& V, const std::vector<int> Neighbours, const std::vector<int> NeighboursSend, const int* RecvOffset, const int* SendOffset, const int* Recv, const int* Send)
```

Итак, функция *Global* использовалась лишь единожды для формирования *L2G*. Схему расстановки гало индексов я придумал из головы, а когда посмотрел на схемы, было уже поздно, поэтому примерная схема представлена на рисунке 1.

Local2Global нетрудно вычисляет *L2G* и *Part* массивы. А *Global2Local* ещё более просто вычисляет *G2L*. Всё это является подготовительным процессом перед началом генерации *IA, JA*. Оперируя обоими массивами (*L2G, G2L*) нетрудно адаптировать *Generate* и *Fill* под MPI реализацию.

Со схемой обмена *Com* особо повозиться тоже не пришлось. Методом пристального вглядывания в свою схему расстановки локальных индексов внезапно осознал, что *RecvFromProcess* и *SendToProcess* не нуждаются в сортировке, но всё ещё – в удалении дубликатов.

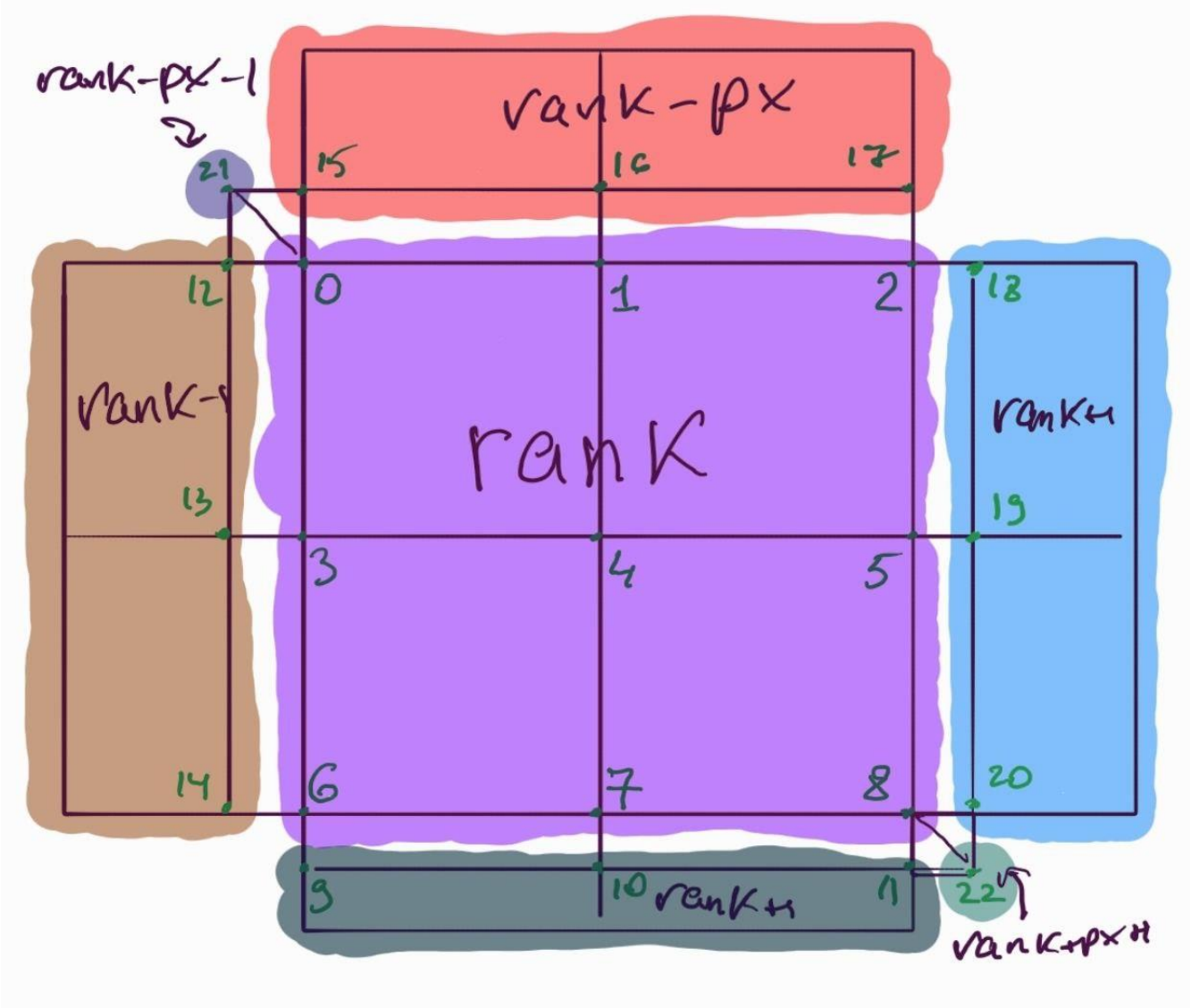

Рисунок 1 – Пример расстановки локальных индексов

Update в целом повторяет идею, подсмотренную из методички, учитывающую особенности *Com*, о которых рассказано выше. Была реализована минимальная версия апдейта перед *SpMV*, но я и ею горжусь!

Axpy и *SpMV* никак не изменились по сравнению с последовательным вариантом.

Для проверки работоспособности кода вычисляются следующие контрольные величины: сумма элементов вектора, l2 норма вектора, min значение вектора.

Для компиляции программы необходимо иметь компилятор g++ (устанавливается следующей командой: `sudo apt install build-essential`) и среду MPI (`sudo apt install mpich`).

Компиляция происходит с помощью следующей команды (при терминале, открытом в папке с файлом программы):

module load gcc/gcc-12.2
module load mpi/latest
mpicxx mpi.cpp -o mpi
sbatch mpi.sb

```
#!/bin/bash
#SBATCH --time=300:00
#SBATCH --output=prog_%j.out
#SBATCH --error=prog_%j.err
#SBATCH --nodelist="vmnode39 vmnode37 vmnode38 vmnode42 vmnode44 vmnode45"
#SBATCH --nodes=6
#SBATCH --ntasks-per-node=6
#SBATCH --ntasks=36
mpirun ./mpi 1000 1000 4 5 6 6
mpirun ./mpi 1000 10000 4 5 6 6
mpirun ./mpi 10000 10000 4 5 6 6
```

Рисунок 2 – batch-файлик для запуска на кластере

2 Результат вычислительных экспериментов

Согласно заданию, были произведены замеры ускорения работы вспомогательных функции, солвера и заполнителей матриц при разном размере массивов. График зависимости продемонстрирован на рисунке 2. При вычислении было проведено усреднение: для вспомогательных функций по 50 запускам, для солвера – по 10.

1. Для всех этапов, включая генерацию и заполнение (то есть этапы инициализации), для фиксированного числа процессов P=18 (на двух вычислительных узлах) был продемонстрирован линейный рост стоимости. Это продемонстрировано графиком на рисунке 3.

batch файл для данного случая выглядел примерно следующим образом:

```
#!/bin/bash
#SBATCH --time=300:00
#SBATCH --ntasks=18
#SBATCH --nodelist="vmnode42 vmnode45"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=9
#SBATCH --output=prog_%j.out
mpirun ./mpi 1000 100 4 5 6 3
mpirun ./mpi 1000 1000 4 5 6 3
mpirun ./mpi 10000 1000 4 5 6 3
mpirun ./mpi 10000 10000 4 5 6 3
```

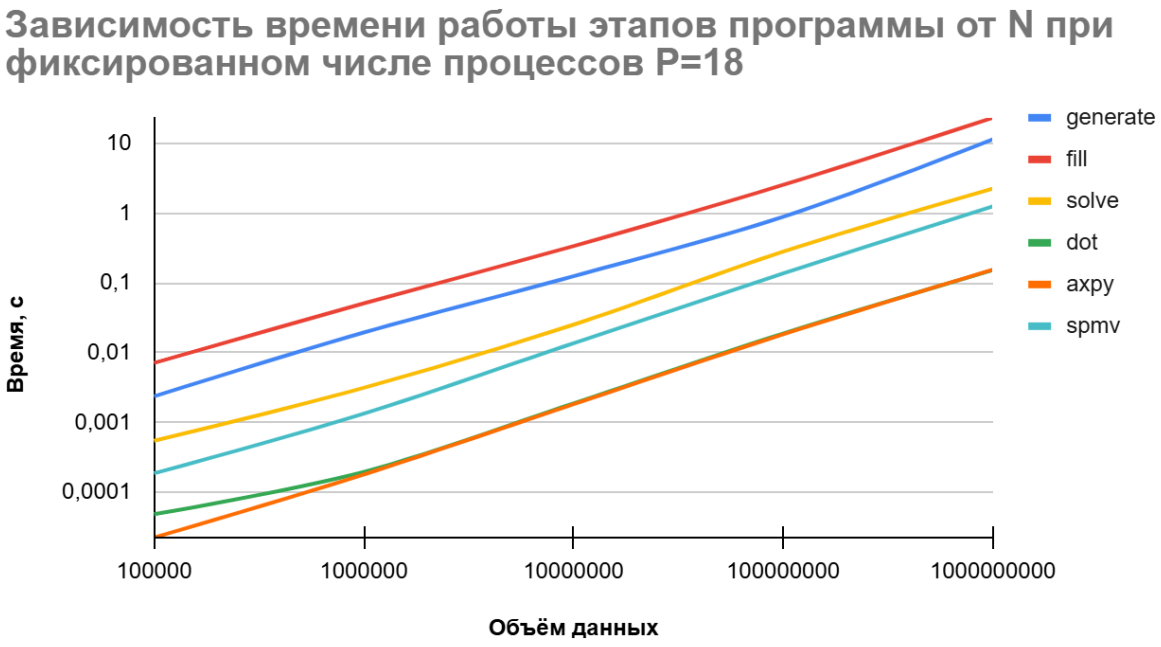

Рисунок 3 – График зависимости времени работы MPI программы от объёма данных N при фиксированном числе процессов (P=18)

2. Сравнение MPI с OpenMP на многоядерном процессоре.

Для каждой из трех базовых операций (кернелов) для этапов заполнения и для всего алгоритма солвера было проведено сравнение ускорения на разном числе ядер при $N=10^8$, полученные в MPI и OpenMP режиме $\left(S = \frac{T_{OpenMP}}{T_{MPI}}\right)$, что продемонстрировано на рисунке 4 и таблицей 1, а также проведена оценка параллельной эффективности MPI алгоритма, что продемонстрировано рисунком 5 и таблицей 2.

Таблица 1 – Ускорение MPI программы над OpenMP версией

	generate	fill	solve	dot	axpy	spmv
4	1,105073334	1,013927769	1,145627111	1,031616902	0,9951315797	1,316876319
8	1,765261077	1,109362833	1,052666133	0,831749821	0,8630213378	1,298659315
12	2,372125657	1,11281029	1,186217569	0,8275271193	0,8432652289	1,673383672
16	2,812733523	1,312888153	1,273392859	0,8459493599	0,8368603887	1,722234677
20	3,044415943	1,394437075	1,357390791	0,8314201183	0,9038616585	1,838986734
24	3,061957036	1,535229533	1,504835067	0,8368244386	1,013342481	1,998640994
28	3,388482435	1,73248194	1,67921555	0,8725786925	1,107140837	2,220040687
32	3,443687453	1,887505433	1,48734166	0,8870476319	1,149294294	1,85682578
36	2,864066619	2,38309326	1,639227051	0,9240914752	1,023943202	2,216883266

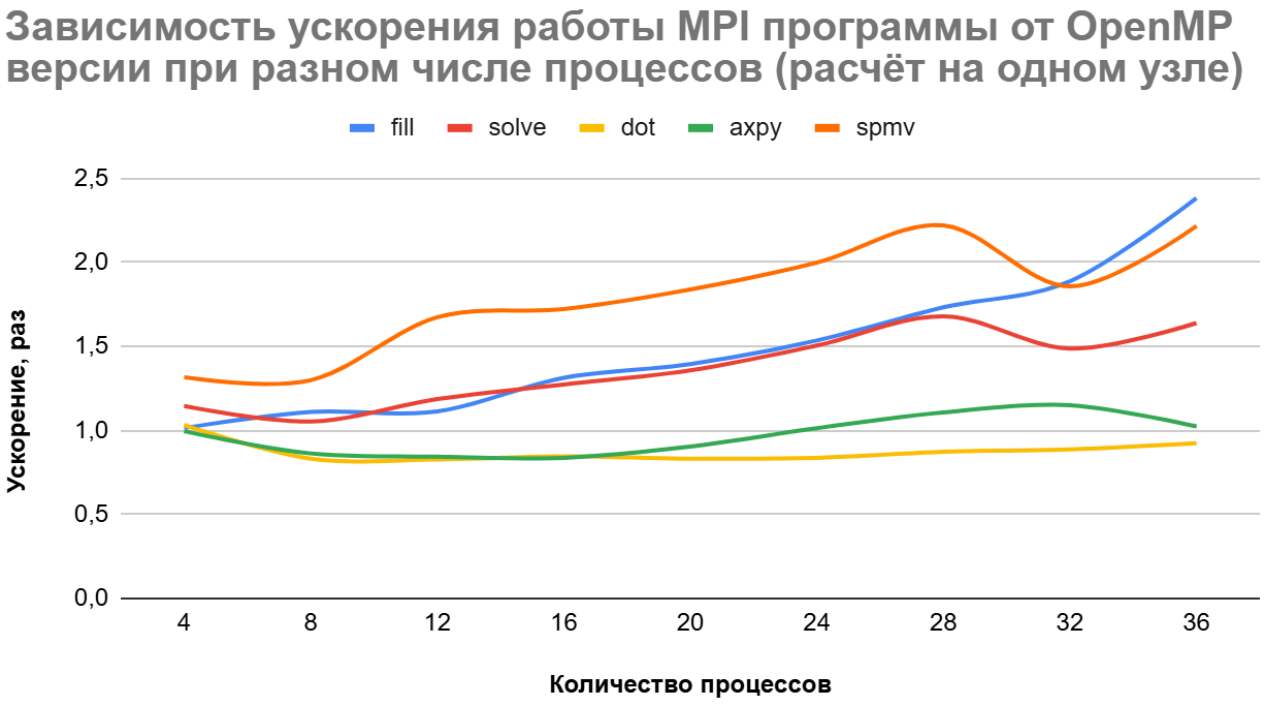

Рисунок 4 – График ускорения работы MPI программы от OpenMP реализацией при разном числе процессов ($N=10^8$)

Как видно из рисунка 4, кернелы axpy и dot сработали в обеих версиях с примерно одинаковой скоростью. SpmV в MPI вариации считается значительно лучше, но справедливости ради при проведении расчётов не учитывалась необходимость обновления гало компонент.

Таблица 2 – Эффективность MPI алгоритма в сравнении с последовательным

	generate	fill	solve	dot	axpy	spmv
4	30,31982436	88,06025145	98,26021835	97,82168186	94,15696052	96,54874605
8	24,74020688	87,52364501	86,49461679	78,36476486	80,03843689	89,32386198
12	21,82411443	84,14550806	85,32903667	77,47991428	79,36754805	89,27845742
16	19,30298162	86,87485606	84,98538883	77,36232349	76,80939583	87,33770487
20	16,64097113	82,57910262	83,16509786	77,69704142	77,59805316	88,93074238
24	14,04858599	80,84525269	77,5581141	71,18168492	69,93715591	82,25891149
28	13,20194825	78,99532095	74,86246195	70,96809063	66,11658538	81,1391426
32	11,75807462	78,01458062	70,92332629	69,45690443	61,41868431	78,43924426
36	8,659626965	76,72582526	65,64935591	68,82230357	56,54897681	77,62157674

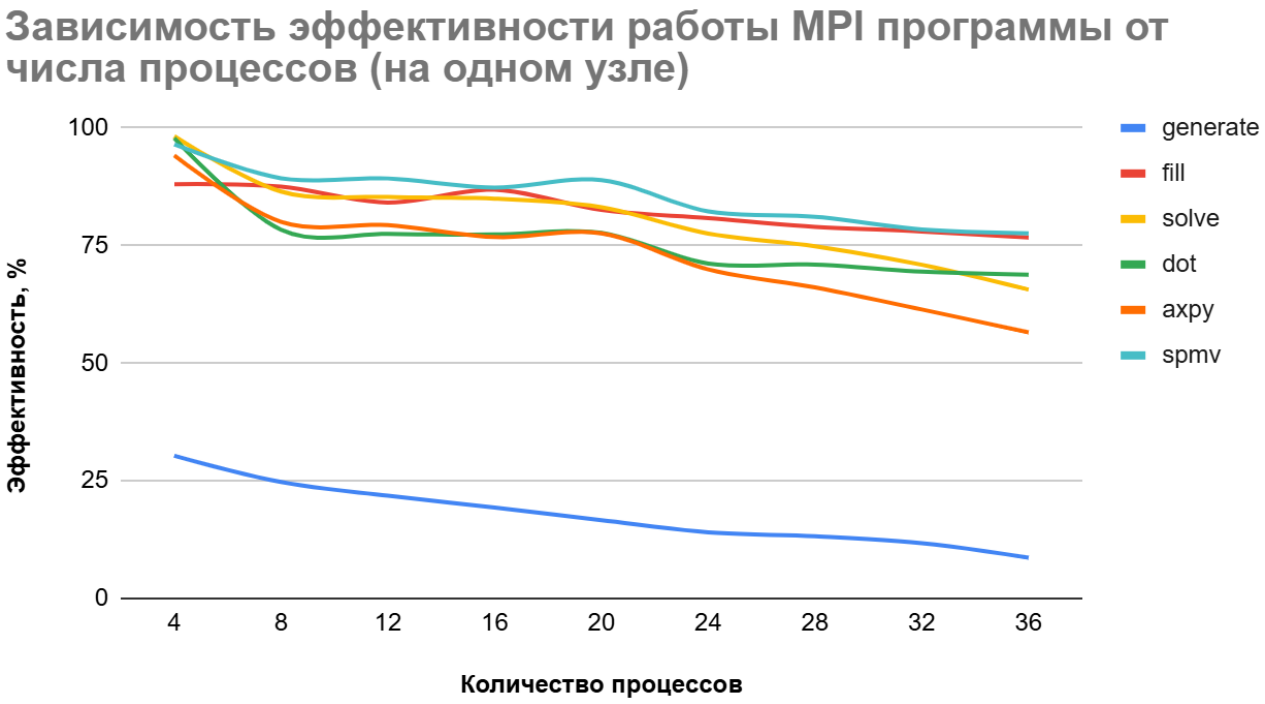

Рисунок 5 – График эффективности работы MPI программы при разном числе процессов ($N=10^8$)

Поведение рисунка 5 отражает реальную ситуацию: действительно, эффективность параллельного алгоритма постепенно снижается с ростом числа процессов, начиная около 100% эффективности в случае кёрнелов и постепенно снижаясь.

Низкую эффективность этапа генерации объясняю очень большим блоком последовательных вычислений: сначала *L2G* и прочих вспомогательных массивов, потом самих *JA-IA*. В OpenMP в своё время вообще не удалось добиться параллелизации алгоритма по этой причине. Также, из-за наличия гало индексов, декомпозиция по данным не позволяет просто считать, что каждый процесс считает N/P элементов, дьявол кроется в деталях.

batch-файлик для данного задания выглядит следующим образом:

```
#!/bin/bash
#SBATCH --time=300:00
#SBATCH --nodelist="vmnode45"
#SBATCH --nodes=1
#SBATCH --output=prog_%j.out
#SBATCH --error=prog_%j.err
#SBATCH --ntasks=32
mpirun ./mpi 10000 10000 4 5 8 4
```

3. Измерение MPI ускорения для различных сеток общее число ячеек N порядка 10^6, 10^7,10^8 для этапов генерации, заполнения, и каждой из 3-х базовых операций и для всего алгоритма солвера: при фиксированном числе N варьируется число процессов и измеряется параллельное ускорение. Программа запускалась на разном числе узлов: по мере возможности от двух до 4. Графики и таблицы ускорения для N порядка 10^6, 10^7,10^8 продемонстрированы на рисунках 6, 7, 8 и таблицами 3, 4, 5 соответственно.

Таблица 3 – Ускорение MPI программы при N = 10^6

	generate	fill	solve	dot	axpy	spmv
4	0,901967908	2,105771674	2,059914098	1,885225913	1,991239248	2,102750146
8	0,8914689249	2,946732517	3,968576516	3,855829253	4,220459149	4,234406913
12	1,032472708	4,174837388	6,298919368	6,238044554	6,742718447	7,488746874
16	2,085252327	7,854188167	9,555611602	8,140083979	9,357035928	10,8655513
20	1,420425288	6,686432017	10,50110865	11,47618397	12,5260521	14,06680585
24	1,559463987	7,362467917	12,23377462	13,57850216	16,84770889	17,57548093
28	1,664636531	8,098267026	13,06032403	14,06344866	17,21900826	17,48426857
32	1,904737549	9,776860154	14,93417422	16,53654856	21,47938144	21,36504162
36	1,666871818	10,16654853	15,62391753	15,7510625	18,54747774	18,46027397

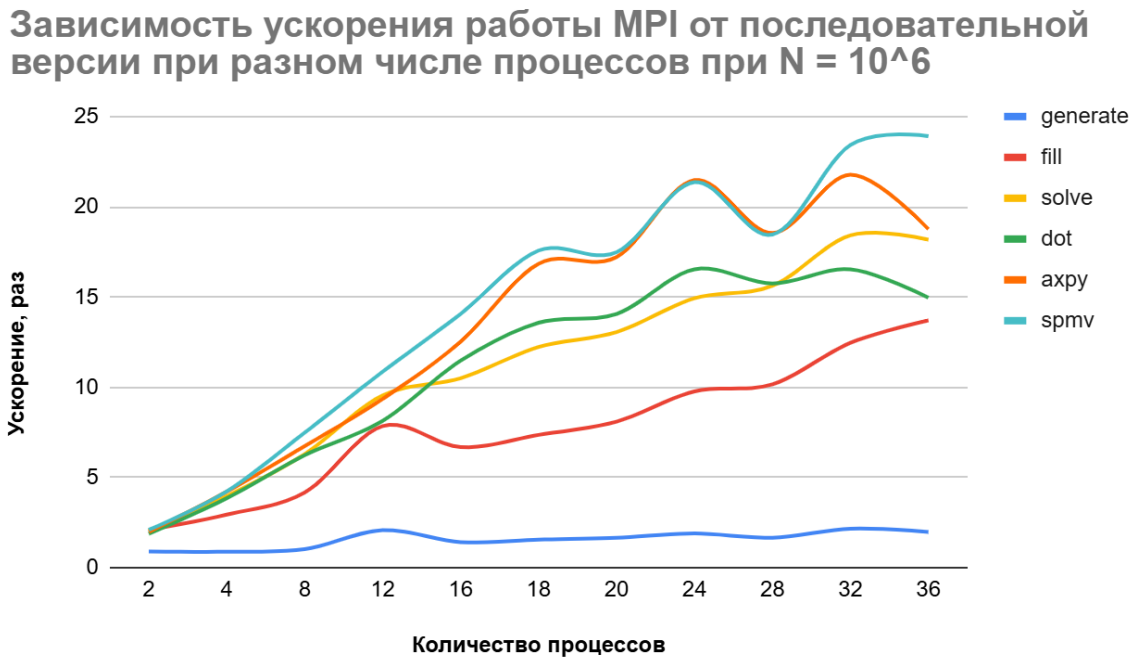

Рисунок 6 – График ускорения работы MPI программы при разном числе процессов (N=10^6)

Таблица 4 – Ускорение MPI программы при $N = 10^7$

	generate	fill	solve	dot	axpy	spmv
4	0,7752086762	1,982174788	2,013051858	2,056693645	2,091743638	2,059141299
8	1,195861079	3,617020406	4,002621467	4,13395683	4,265873016	3,968328921
12	1,748230202	7,151498482	6,950947053	6,81785677	6,869829232	7,627531695
16	1,59159821	7,059074821	10,22260762	9,229543527	9,571264962	10,59807757
20	1,888740945	13,71239559	13,44897237	12,41837802	12,66746528	14,22636027
24	1,918312149	10,49828699	16,59416874	17,10535621	17,89191989	17,25721955
28	1,961383034	11,64661439	17,10867609	17,60804799	18,47895863	17,5
32	2,135145548	13,94552216	21,10753942	22,41958462	23,79454445	22,42771287
36	2,235322809	16,21922357	23,23467579	25,98816156	27,46766327	26,67824826

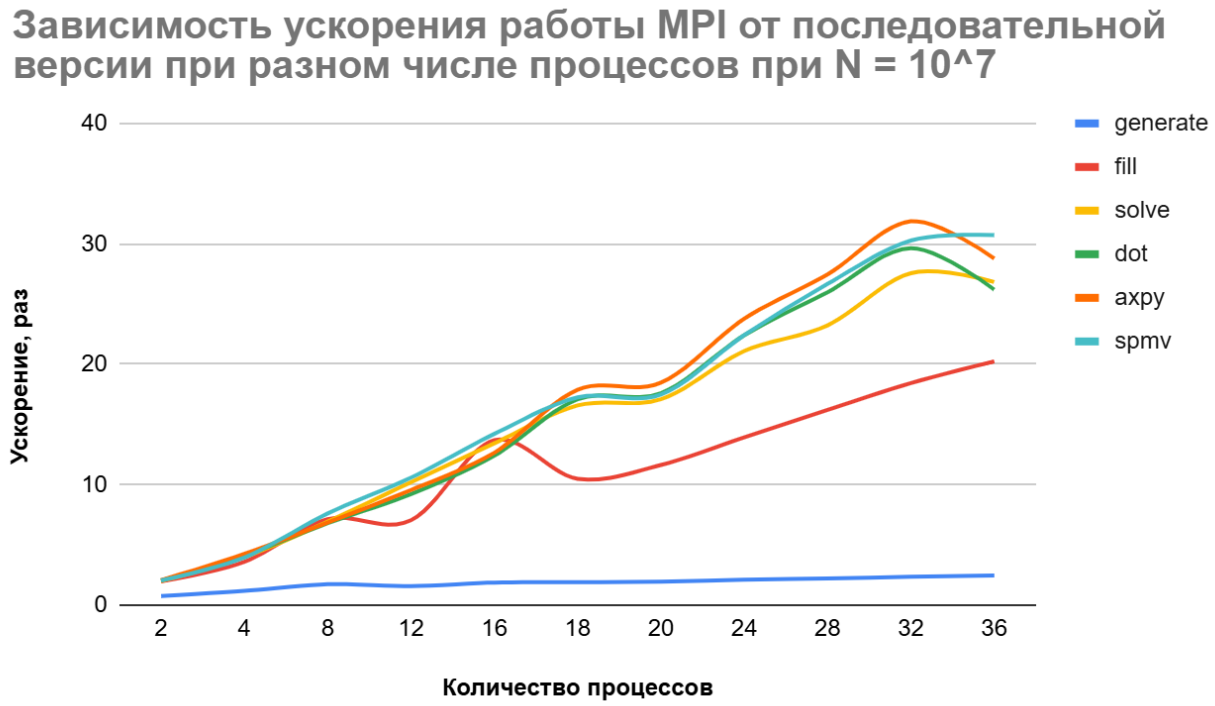

Рисунок 7 – График ускорения работы MPI программы при разном числе процессов ($N=10^7$)

Таблица 5 – Ускорение MPI программы при $N = 10^8$

	generate	fill	solve	dot	axpy	spmv
4	0,672147774	1,850082869	2,00134894	2,029264131	2,103252521	1,970483686
8	1,296564587	3,667687383	4,073643837	3,940295999	4,136502971	4,077764254
12	2,016652028	7,103422137	7,20359145	6,579809783	6,769313116	7,475017849
16	2,126685191	10,45030477	10,50211677	9,409657031	9,671461926	10,67471389
20	2,537439554	13,84754483	13,62691555	12,46776429	12,70854774	14,29252288
24	3,33172108	15,66074994	17,32787703	15,84926613	17,52704802	17,47392973
28	2,674687454	17,35412372	18,51015845	18,21849766	19,81643807	19,28168817
32	2,868389606	20,64379424	23,02849399	22,88712264	23,62819957	23,15138336
36	3,014072256	23,48017883	26,33002287	25,47592255	25,37422944	26,55042695

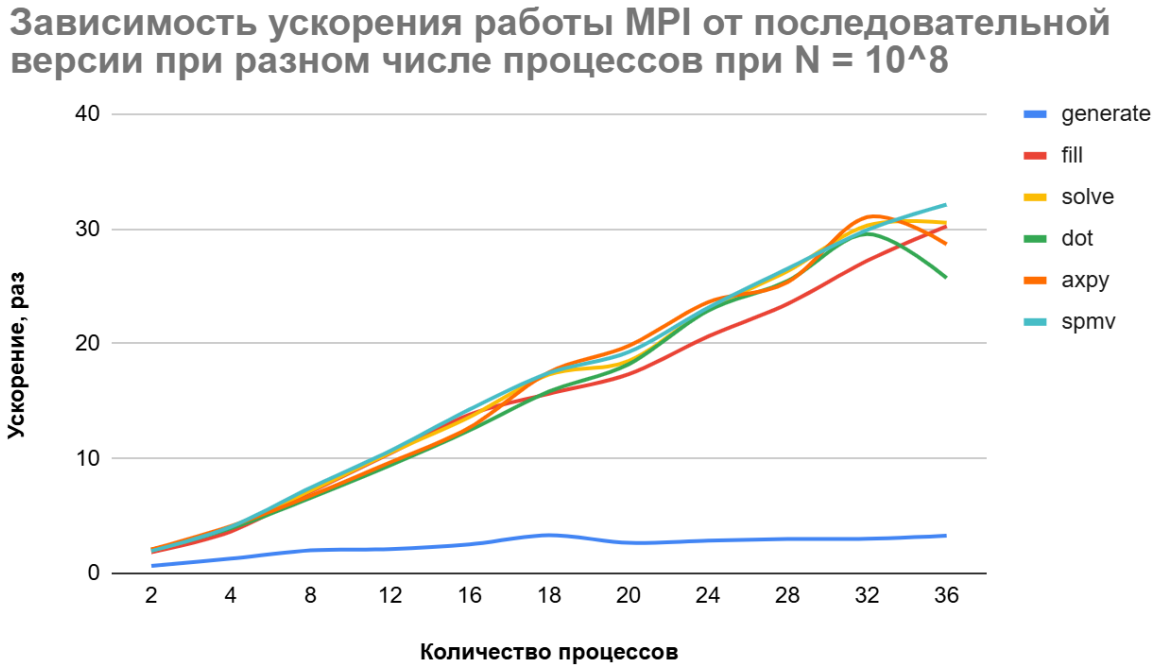

Рисунок 8 – График ускорения работы MPI программы при разном числе процессов ($N=10^8$)

Рисунок 8 наиболее показательно выражает общую мысль: основная тусовка из кернелов и ко неплохо так ускоряется с ростом числа параллельных процессов, что также видно и по рисунку 5. *Generate* же в эту тусовку явно не приглашали. Но по крайней мере почти всюду даже он приобретает ускорение, о котором в OpenMP версии даже не слышали.

Batch файлик для такого случая запусков выглядел как на рисунке 2.