



**Практикум по учебному курсу
«Суперкомпьютеры и параллельная обработка данных»**

Задание

Разработка параллельной версии программы для вычисления
определенного интеграла с использованием метода Симпсона

**Отчет
о выполненном задании**

студента 320 учебной группы факультета ВМК МГУ

Москаленко Андрея Викторовича

Москва, 2019 г.

Содержание

1. Постановка задачи	2
2. Описание метода	2
3. Реализация алгоритма.....	3
3.1 Последовательная версия.....	3
3.2 Параллельная версия (OpenMP)	3
3.3 Параллельная версия (MPI)	4
4. Тестирование	5
4.1 Сравнение используемых функций	5
4.2 Результаты тестирований	6
5. Визуализации	7
5.1 Технология OpenMP.....	7
5.2 Технология MPI	9
6. Анализ результатов	13
7. Выводы	14

1. Постановка задачи

Ставится задача нахождения численного значения определенного интеграла функции $f(x)$ на отрезке $[a, b]$ с использованием метода Симпсона.

Требуется:

- Реализовать последовательную и параллельную версию программы
- Протестировать полученные программы на суперкомпьютерах Polus и Blue Gene
- Сравнить эффективность, проанализировать масштабируемость

2. Описание метода

Для удобства разобьём отрезок $[a, b]$ на $N = 2n$ элементарных отрезков. Тогда значение исходного интеграла является суммой результатов интегрирования на этих отрезках:

$$\int_a^b f(x) \approx \frac{h}{3} \left[f(x_0) + 2 \sum_{j=1}^{n-1} f(x_{2j}) + 4 \sum_{j=1}^n f(x_{2j-1}) + f(x_N) \right]$$

$$\text{где } h = \frac{b-a}{N}, x_j = a + jh$$

Полученную формулу можно переписать в виде:

$$\int_a^b f(x) \approx \frac{h}{3} \sum_{k=1,2}^{N-1} [f(x_{k-1}) + 4f(x_k) + f(x_{k+1})]$$

где запись $k = 1,2$ означает, что k меняется от единицы с шагом 2

3. Реализация

Были реализованы три версии программы:

- 1) Наивная реализация последовательной программы (baseline)
- 2) Последовательная версия
- 3) Параллельная версия с использованием OpenMP
- 4) Параллельная версия с использованием MPI

3.1 Последовательная версия (наивная реализация)

```
double integral(double (*f)(double), const double a, const double b,
               const long long n) {
    double h = (b - a) / (2 * n);
    double sum = 0.0;
    long long N = 2 * n;

    for (long long k = 1; k <= N - 1; k += 2) {
        double x_k = a + k * h;
        sum += f(x_k - h) + 4 * f(x_k) + f(x_k + h);
    }
    return sum * h / 3;
}
```

3.2 Последовательная и параллельная версии (отличия выделены цветом)

```
double integral(double (*f)(double), const double a, const double b,
               const long long n) {
    double h = (b - a) / (2 * n);
    double sum = 0.0;
    long long N = 2 * n;
    double sum1 = 0.0;
    double sum2 = 0.0;

#pragma omp parallel for reduction(+:sum1) reduction(+:sum2)
    for (long long j = 1; j < N; j += 2) {
        sum1 += f(a + j * h);
        sum2 += f(a + (j + 1) * h);
    }
    #минус f(b) т.к мы её добавили дважды в sum2
    sum += f(a) - f(b) + 4 * sum1 + 2 * sum2;
    return sum * h / 3;
}
```

С полным кодом программы можно ознакомиться в репозитории – <https://github.com/a-mos/skpod>

3.3 Параллельная версия с использованием MPI

```
double integral(double (*f)(double),
    const double a, const double b, const long long N, const double h) {
    double sum = 0.0;
    double sum1 = 0.0;
    double sum2 = 0.0;
    for (long long j = 1; j < N; j += 2) {
        sum1 += f(a + j * h);
        sum2 += f(a + (j + 1) * h);
    }
    sum += f(a) - f(b) + 4 * sum1 + 2 * sum2;
    return sum * h / 3;
}

int main(int argc, char** argv) {
    int rank, size, mode;
    long long n = 0;
    double a, b, result, total;

    MPI_Init(NULL, NULL);
    double start = MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (rank == 0) {
        sscanf(argv[1], "%d", &mode);
        sscanf(argv[2], "%lf", &a);
        sscanf(argv[3], "%lf", &b);
        sscanf(argv[4], "%lld", &n);
    }

    MPI_Bcast(&mode, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    double h = (b - a) / (2 * n);
    long long my_n = (2 * n) / size;
    double my_a = a + rank * my_n * h;
    double my_b = my_a + my_n * h;

    if (mode == 1) {
        result = integral(f1, my_a, my_b, my_n, h);
    } else if (mode == 2) {
        result = integral(f2, my_a, my_b, my_n, h);
    } else if (mode == 3) {
        result = integral(f3, my_a, my_b, my_n, h);
    }

    MPI_Reduce(&result, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double end = MPI_Wtime();
        printf("TIME: %lf\n", end - start);
    }

    MPI_Finalize();
}
```

С полным кодом программы можно ознакомиться в репозитории – <https://github.com/a-mos/skpod>

4. Тестирование

Для теста были выбраны следующие функции:

$$f_1(x) = x^3$$

$$f_2(x) = \frac{1}{1 + e^{-x}}$$

$$f_3(x) = \sqrt{1 - \sin(x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7)}$$

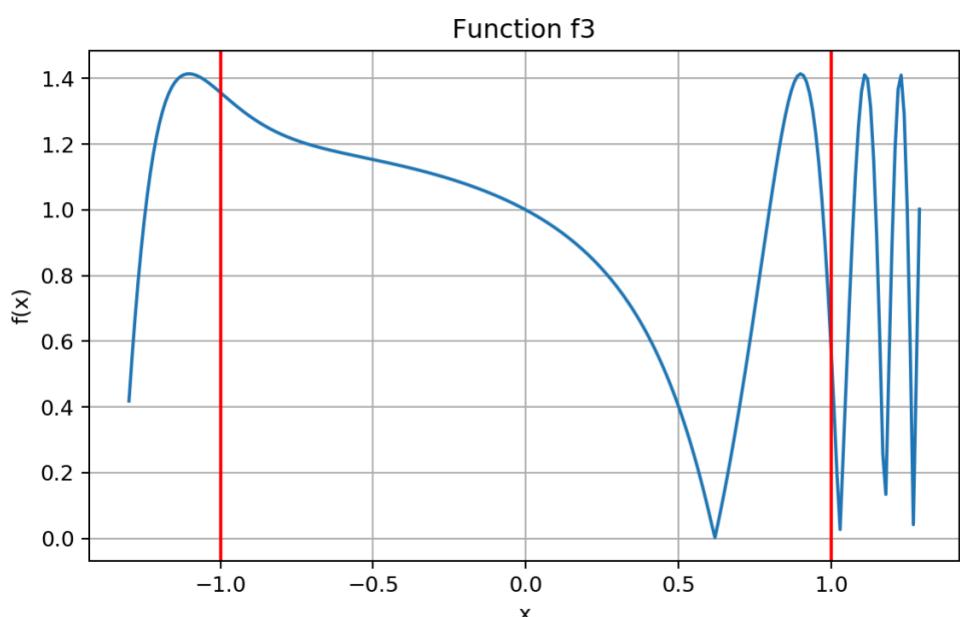
Компиляция производилась с включенной оптимизацией (-O3). Замер времени начался с момента вызова функции и заканчивался в момент возврата. Тестирование проводилось трижды, затем результаты усреднялись.

4.1 Сравнение используемых функций

	1 threads	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads	128 threads
f1	0.701202	0.370331	0.255652	0.157813	0.135874	0.112560	0.141225	0.210439
f2	3.082734	1.574076	0.848379	0.562202	0.423933	0.289902	0.289148	0.374779
f3	10.535222	5.295680	2.805154	1.520768	1.106129	0.808859	0.707183	0.779625

Время исполнения функций в зависимости от числа нитей (Polus)

Заметим, что изменение функции лишь пропорционально влияет на скорость выполнения программы, поэтому в последующих сравнениях остановимся на интегрировании функции $f_3(x)$ на интервале $[-1, 1]$:



4.2 Результаты тестирования

4.2.1 Технология OpenMP (Polus)

	1	2	4	8	16	32	64	96	128
100000000	4.2789	2.11689	1.2932	0.774501	0.578737	0.449088	0.388636	0.345807	0.422473
200000000	8.40787	4.92225	2.18933	1.57918	1.38082	0.88881	0.591938	0.668882	0.616773
300000000	12.9831	6.791	3.56433	2.36775	1.63882	1.34545	0.727472	0.864847	0.818125
400000000	17.6631	8.94617	4.53019	3.16255	2.41023	1.77456	1.02767	1.12305	1.147
500000000	20.3293	10.3987	5.91088	3.68323	2.73115	2.14443	1.21934	1.30253	1.2294
600000000	26.1833	12.6393	7.2789	4.71724	3.2736	2.69151	1.73286	1.83238	1.73742
700000000	30.8109	15.5738	7.7918	5.10263	3.73365	3.10522	1.65454	1.66355	1.72212
800000000	32.5913	19.4902	8.58503	5.38475	5.0262	3.46792	1.96903	1.92851	2.10592
900000000	36.7437	19.6213	10.0643	5.23304	4.83343	3.659	2.25586	1.82512	2.19112
1000000000	40.7387	21.3874	10.8865	6.81372	4.89778	4.17155	2.31908	2.21437	2.40179

Время работы при разном числе нитей и числе интервалов

4.2.2 Технология MPI (BlueGene – сверху и Polus – снизу)

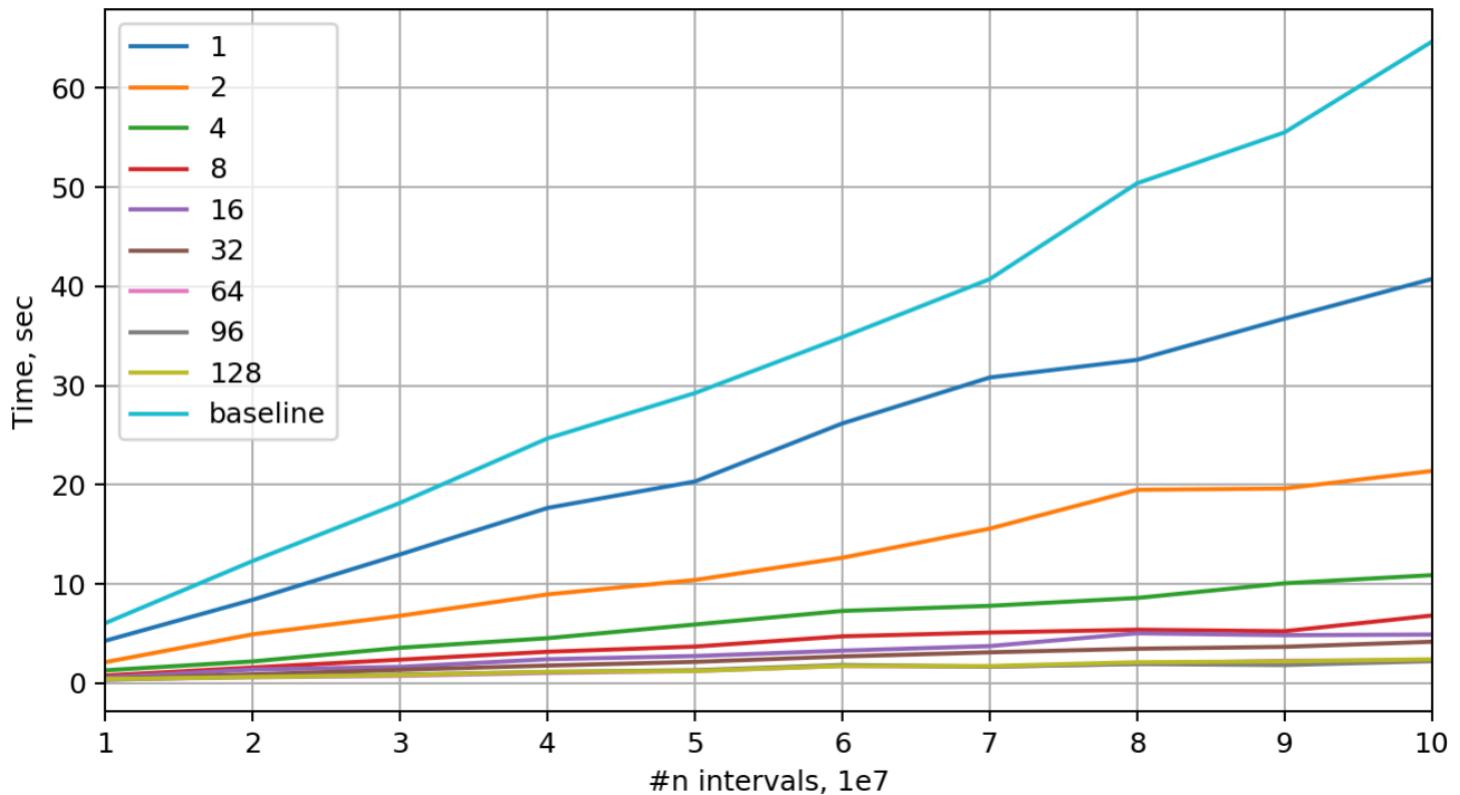
	1	2	4	8	16	32	64	128	256	512	1024
100000000	69.8465	30.4231	14.4188	7.02786	3.51408	1.75718	0.878589	0.4394	0.21972	0.10995	0.055062
200000000	139.69	60.845	28.8362	14.0549	7.0279	3.51426	1.7571	0.878627	0.439319	0.21975	0.109964
300000000	209.541	91.2699	43.2533	21.0817	10.5412	5.27098	2.63544	1.31787	0.658992	0.329522	0.164842
400000000	279.022	121.691	57.6701	28.1084	14.0548	7.02783	3.51385	1.75706	0.878538	0.439386	0.21979
500000000	349.227	152.111	72.0883	35.1348	17.5681	8.78468	4.39251	2.19641	1.09821	0.549144	0.274653
600000000	419.041	182.536	86.5045	42.1629	21.0821	10.5417	5.27096	2.63552	1.31788	0.65896	0.329562
700000000	488.915	212.953	100.921	49.1884	24.5954	12.2985	6.14919	3.07477	1.5374	0.768846	0.384426
800000000	558.028	243.377	115.338	56.2159	28.1092	14.0553	7.0278	3.51406	1.75694	0.878566	0.439361
900000000	628.609	273.801	129.756	63.2437	31.6235	15.8126	7.9063	3.9533	1.97681	0.98853	0.494309
1000000000	698.443	304.219	144.177	70.2702	35.136	17.5692	8.7849	4.39257	2.19628	1.09818	0.549189
	1	2	4	8	16	32	64				
100000000	44.367	22.1635	10.126	5.303	3.06371	1.78749	0.840502				
200000000	90.3582	47.5064	19.7345	11.4103	6.42696	4.13002	1.55968				
300000000	150.079	69.395	33.953	16.7068	8.66409	5.45898	2.21719				
400000000	198.209	82.4864	40.7498	21.416	13.6815	6.817	3.10079				
500000000	244.125	106.217	50.4014	30.8399	15.9959	9.26668	3.67917				
600000000	284.912	134.515	64.2276	35.6692	18.9701	12.0582	4.46643				
700000000	329.12	156.932	71.5672	41.0705	19.2608	13.957	4.78443				
800000000	388.077	160.384	83.7299	41.3814	28.6328	16.5743	5.56435				
900000000	456.899	179.807	101.676	46.856	29.283	18.5759	6.77783				
1000000000	538.339	219.467	106.367	51.1976	34.0679	20.0191	7.15999				

Слева число интервалов, сверху – число процессов

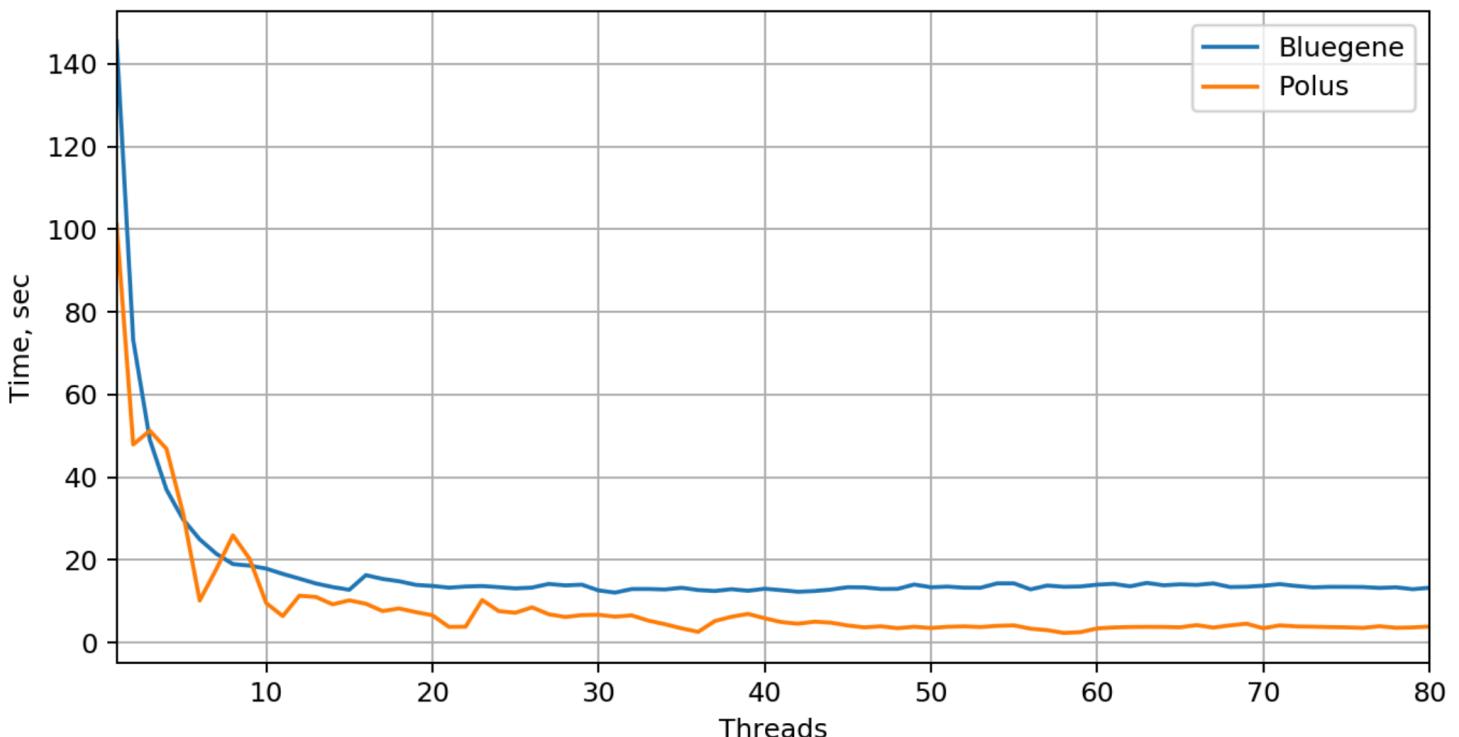
5. Визуализации

5.1 Технология OpenMP

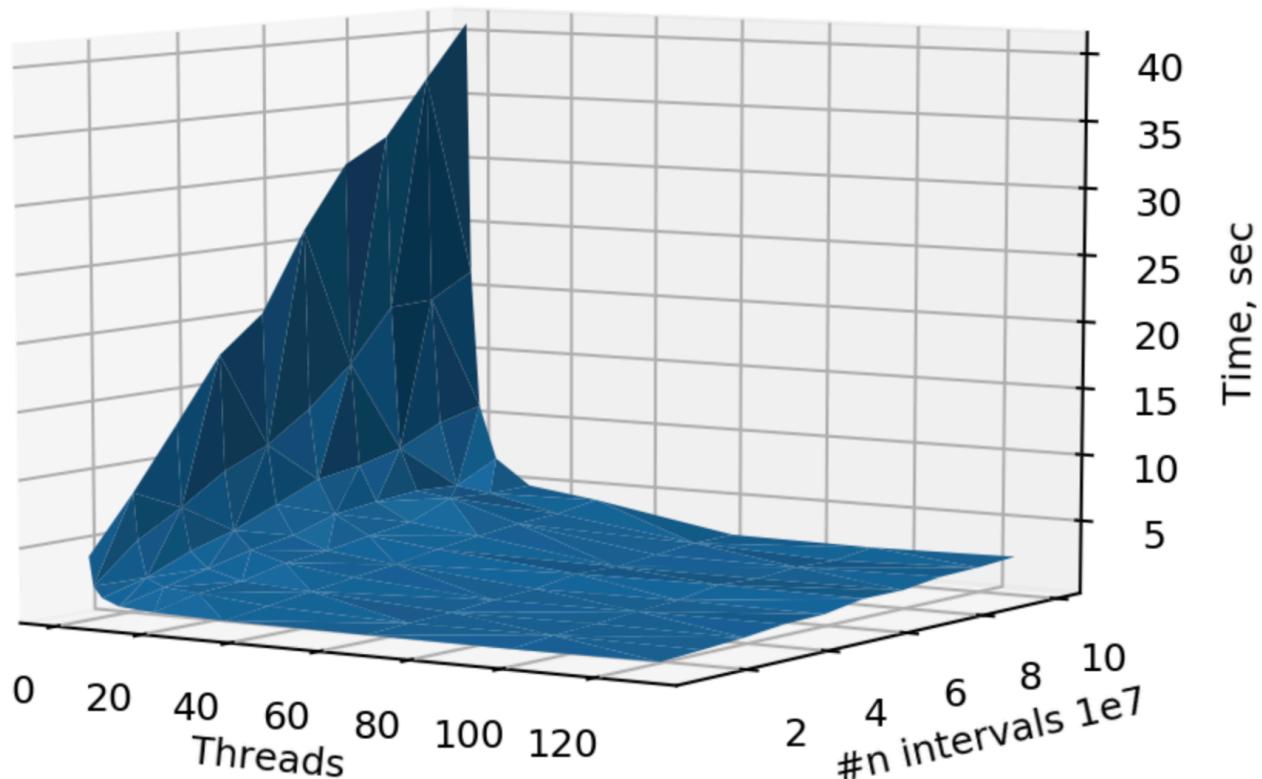
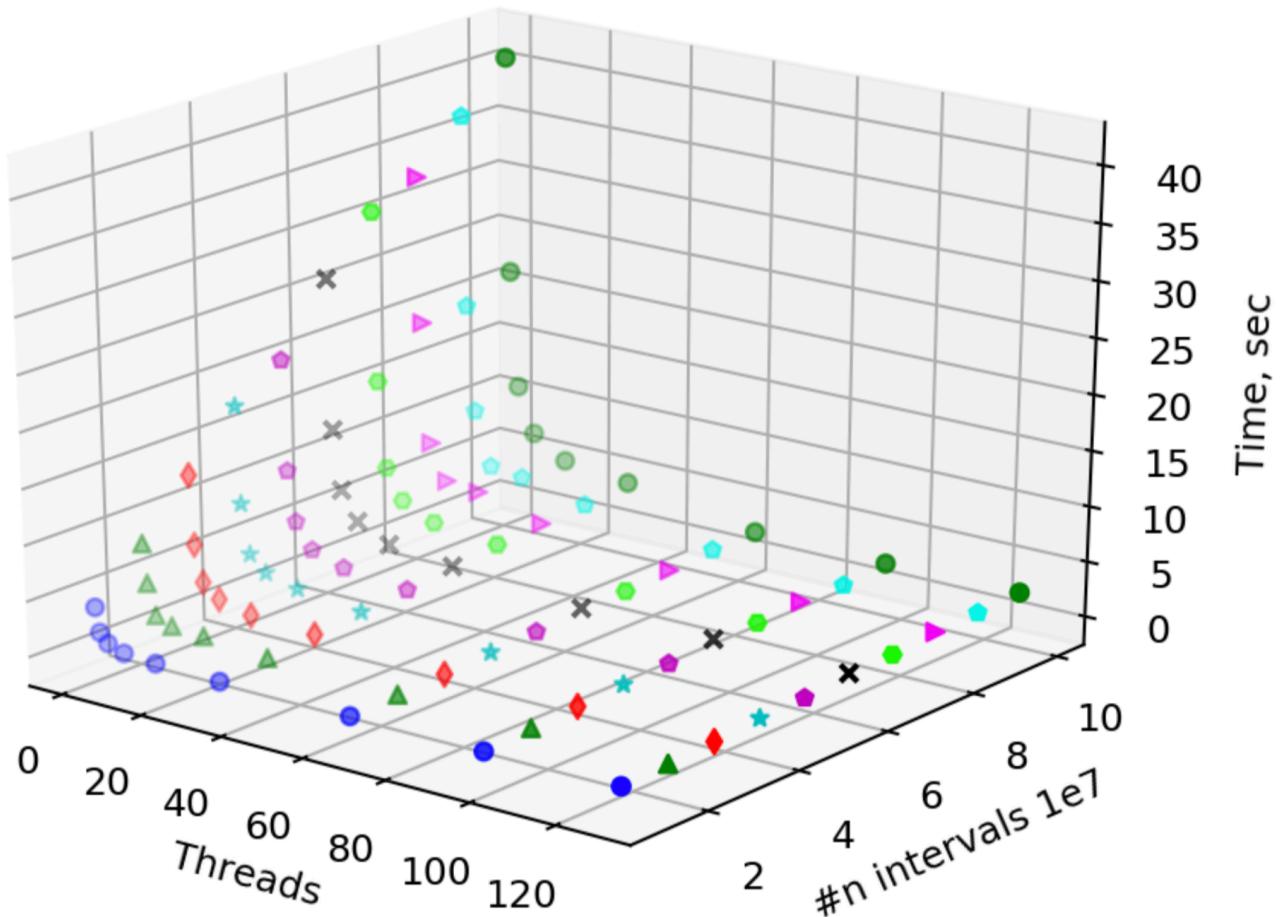
5.1.1 Сравнение скорости работы всех версий программы, включая наивную реализацию (baseline)



5.1.2 Время работы на BlueGene* и Polus ($n = 10^8$)

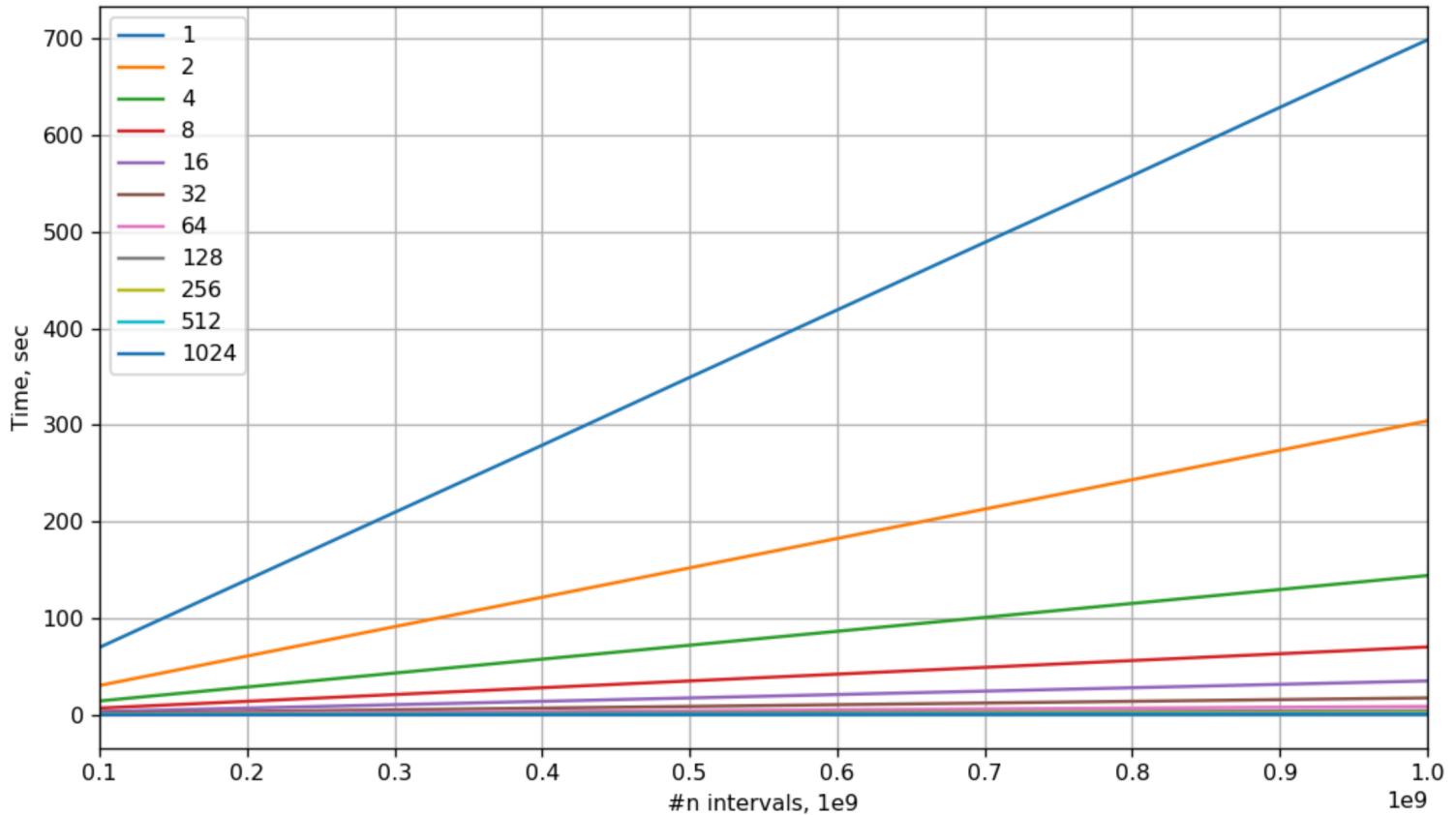


5.1.3 Пространственная визуализация (Polus)

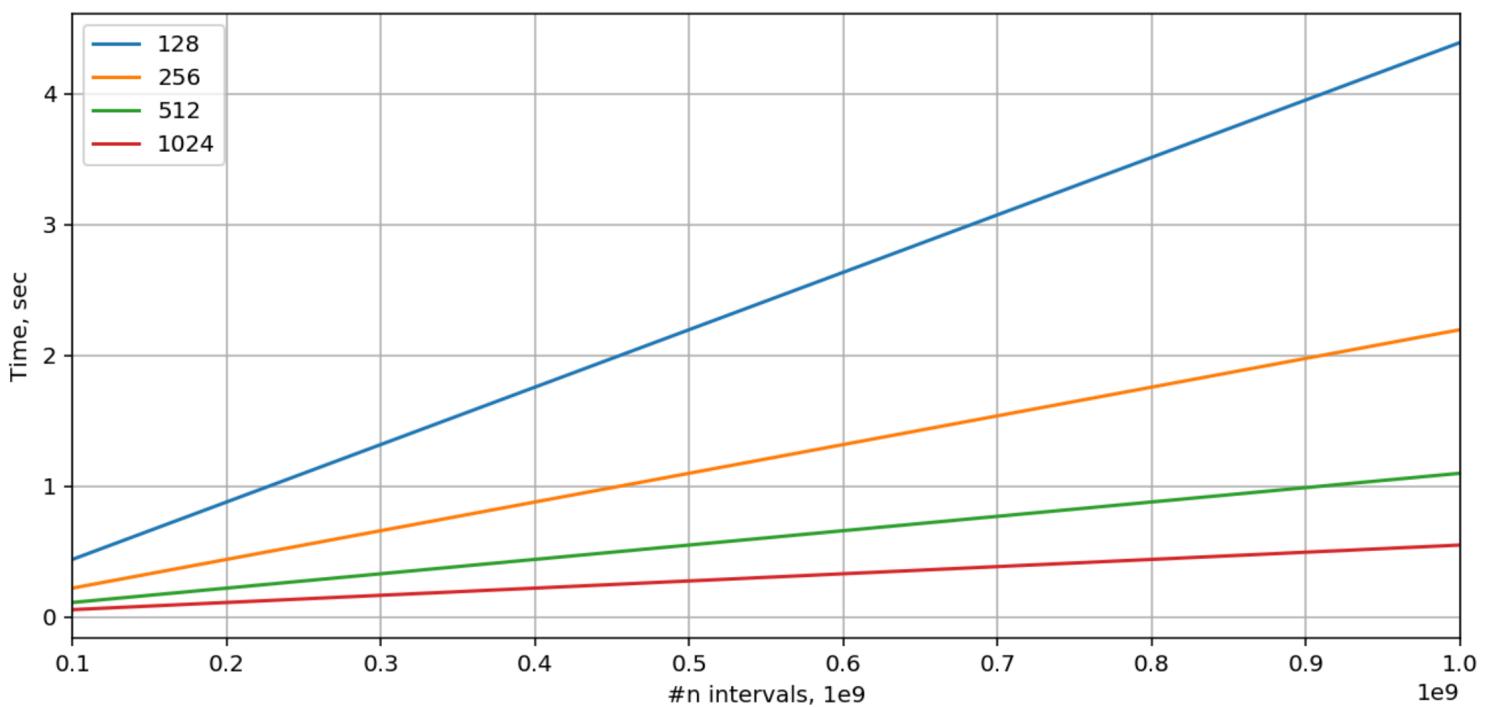


5.2 Технология MPI

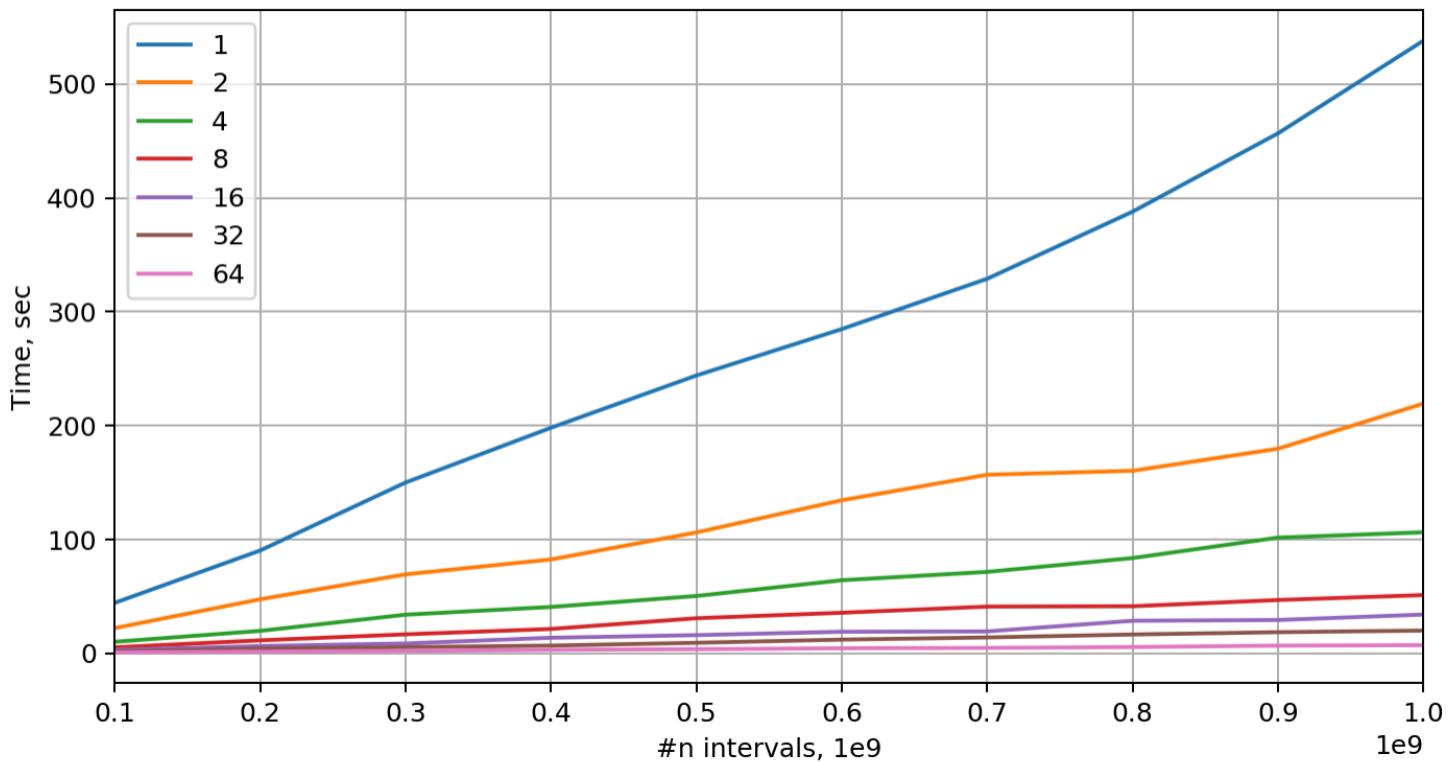
5.2.1 Сравнение скорости работы в зависимости от числа процессов и числа интервалов (BlueGene)



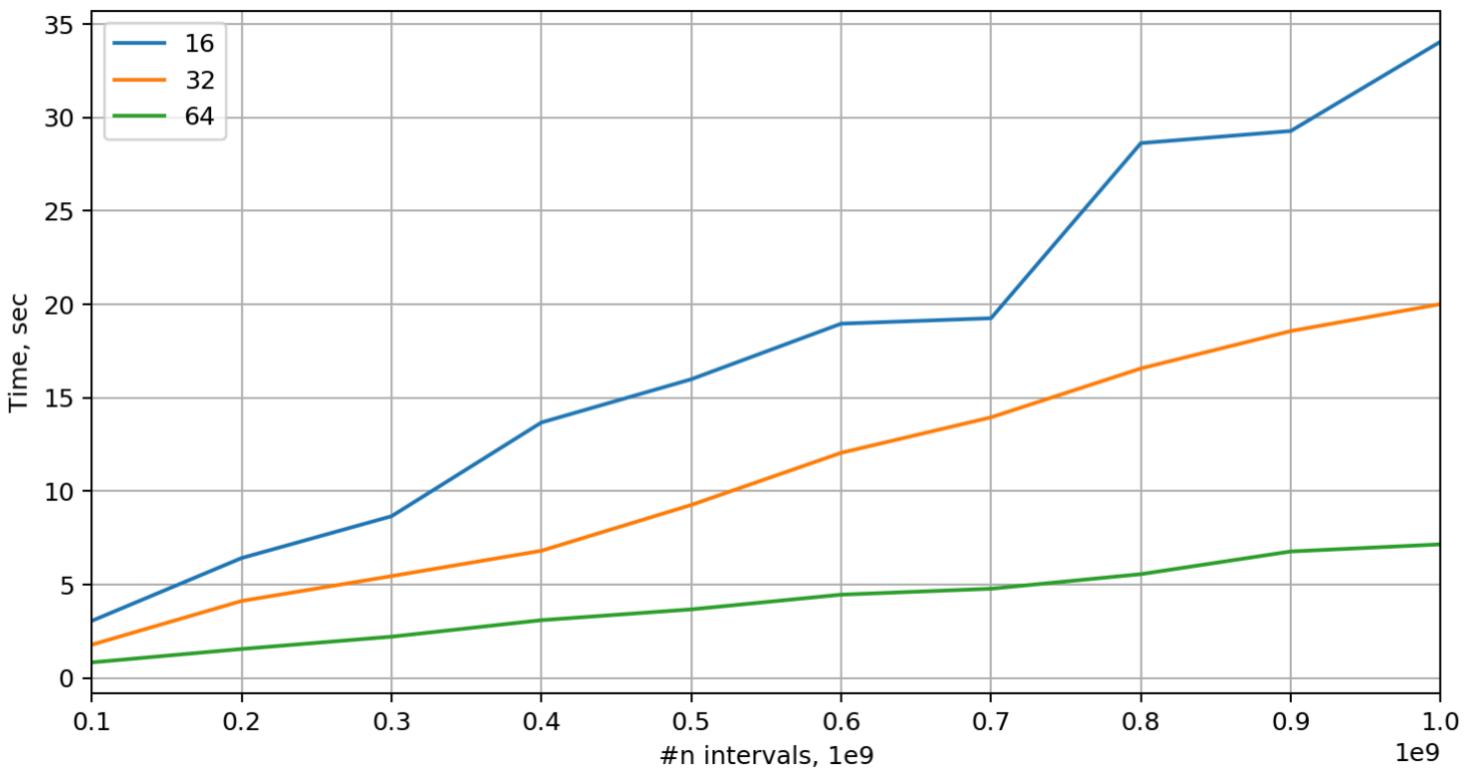
5.2.2 Только запуски с большим количеством процессов



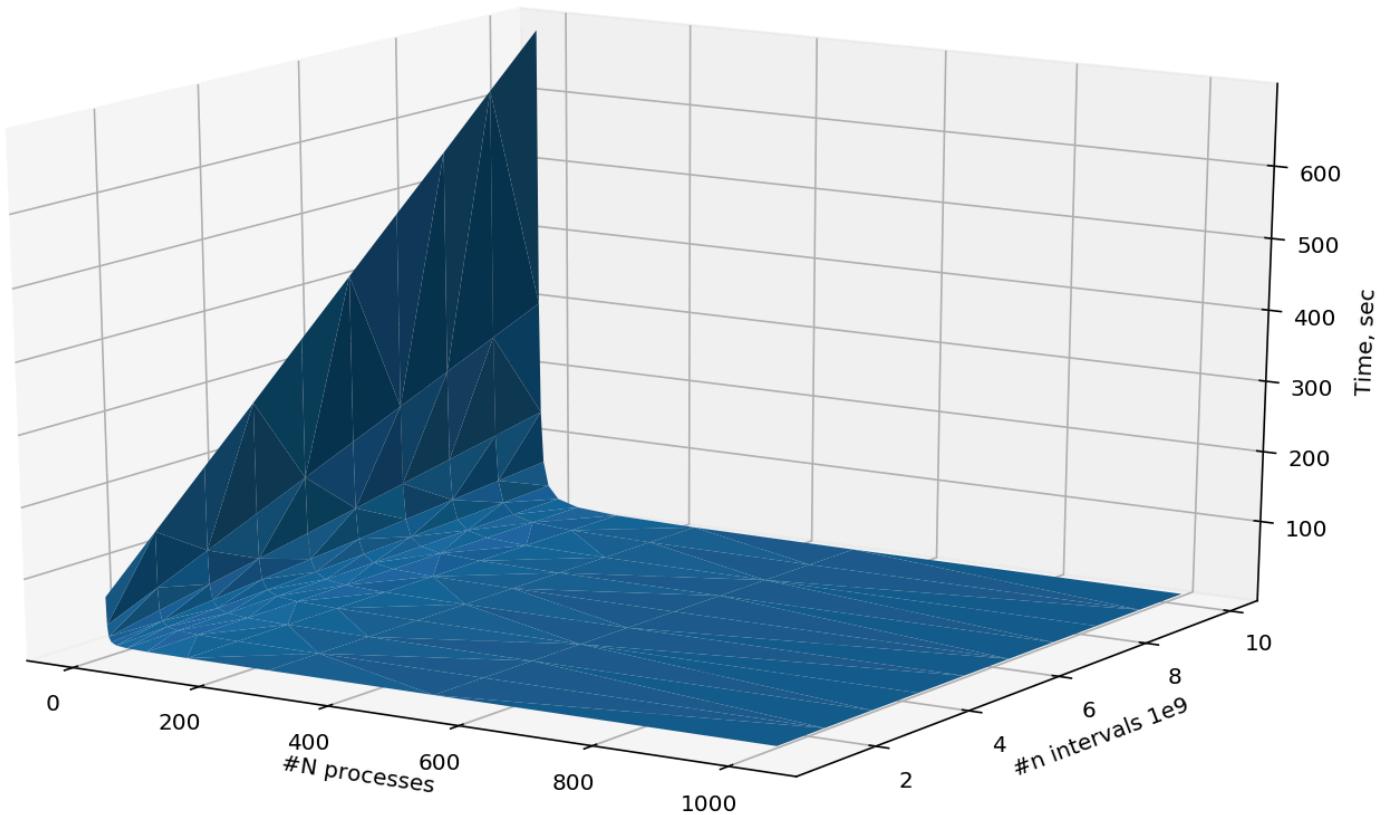
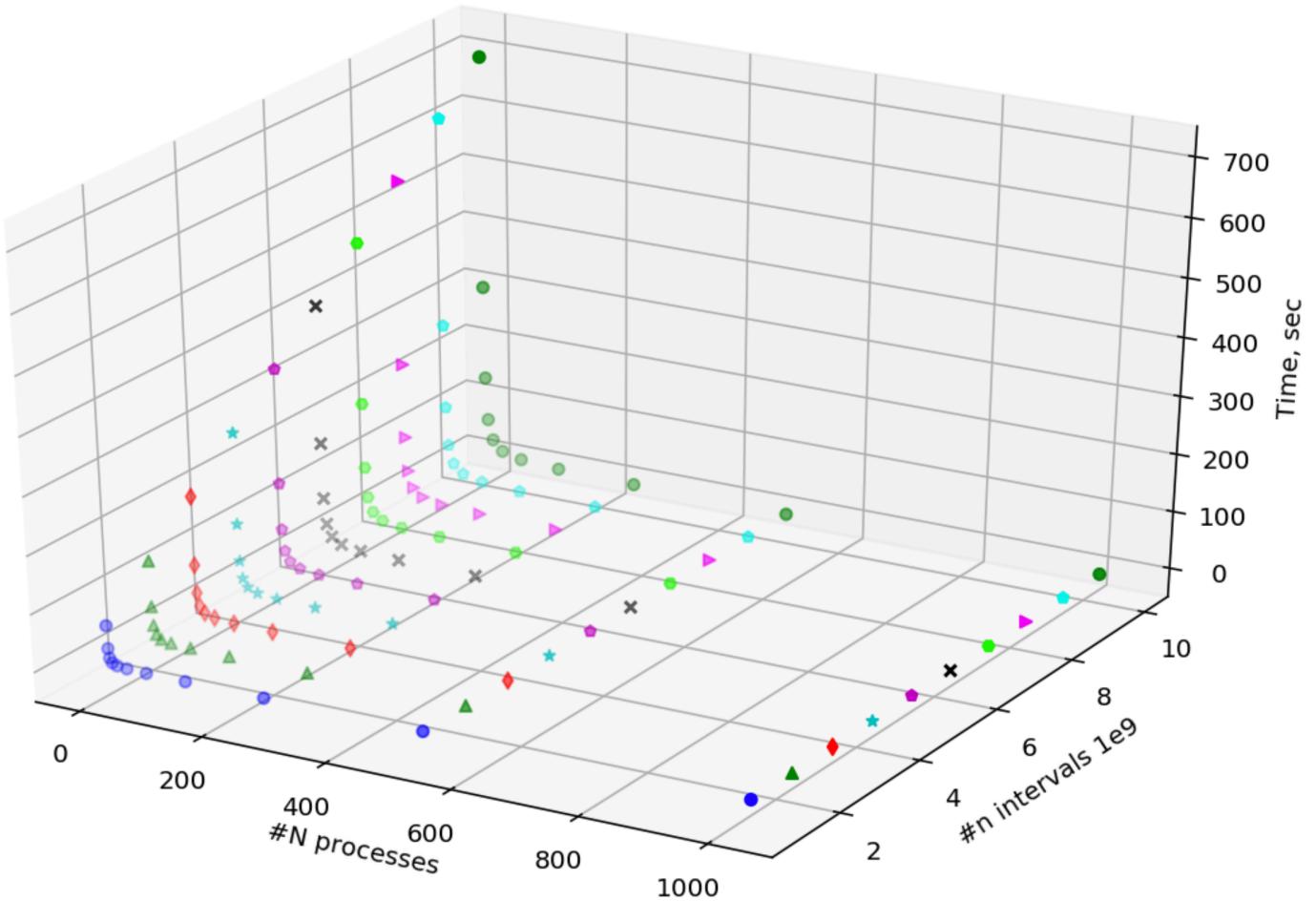
5.2.3 Сравнение скорости работы в зависимости от числа процессов и числа интервалов (Polus)



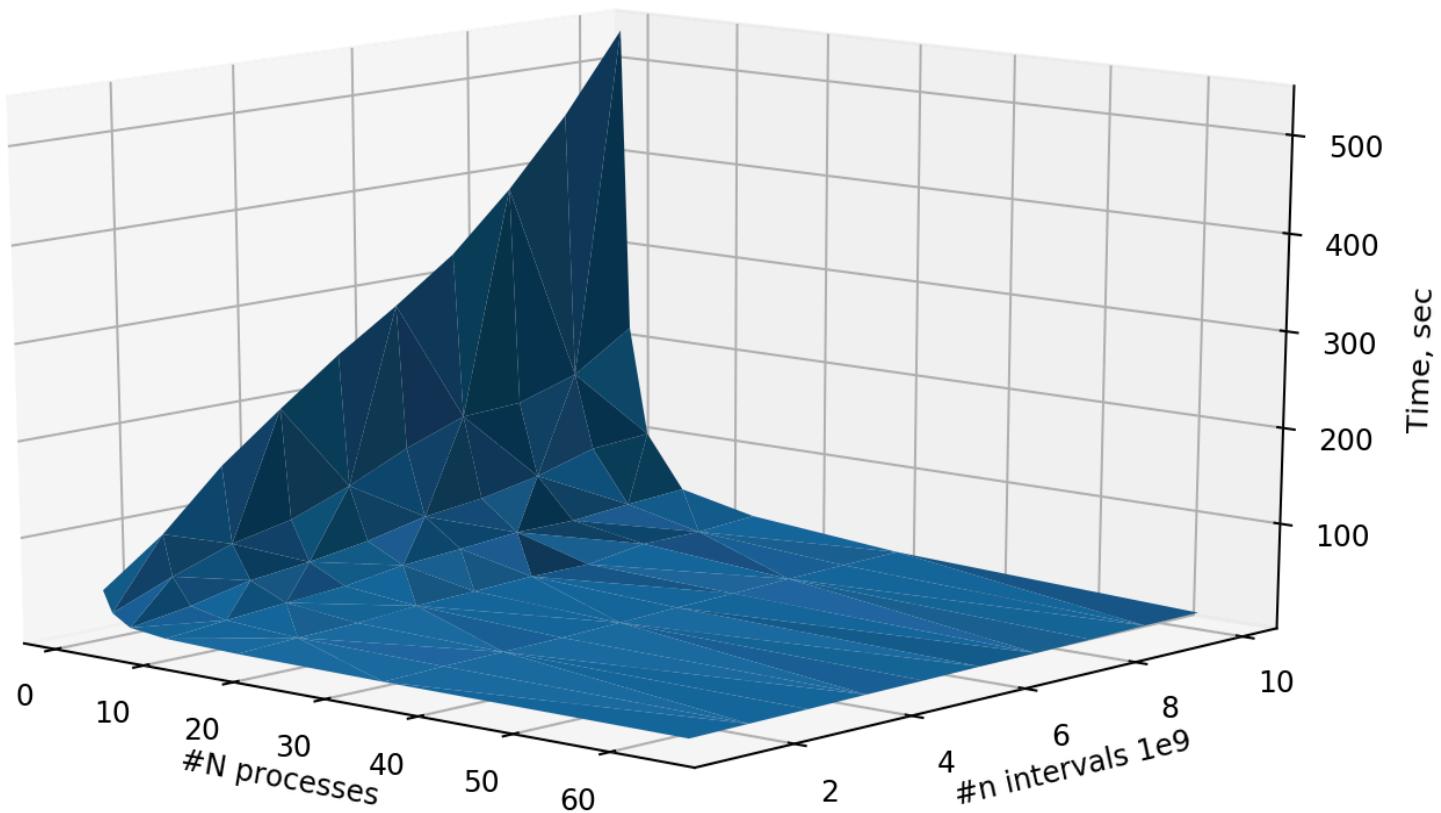
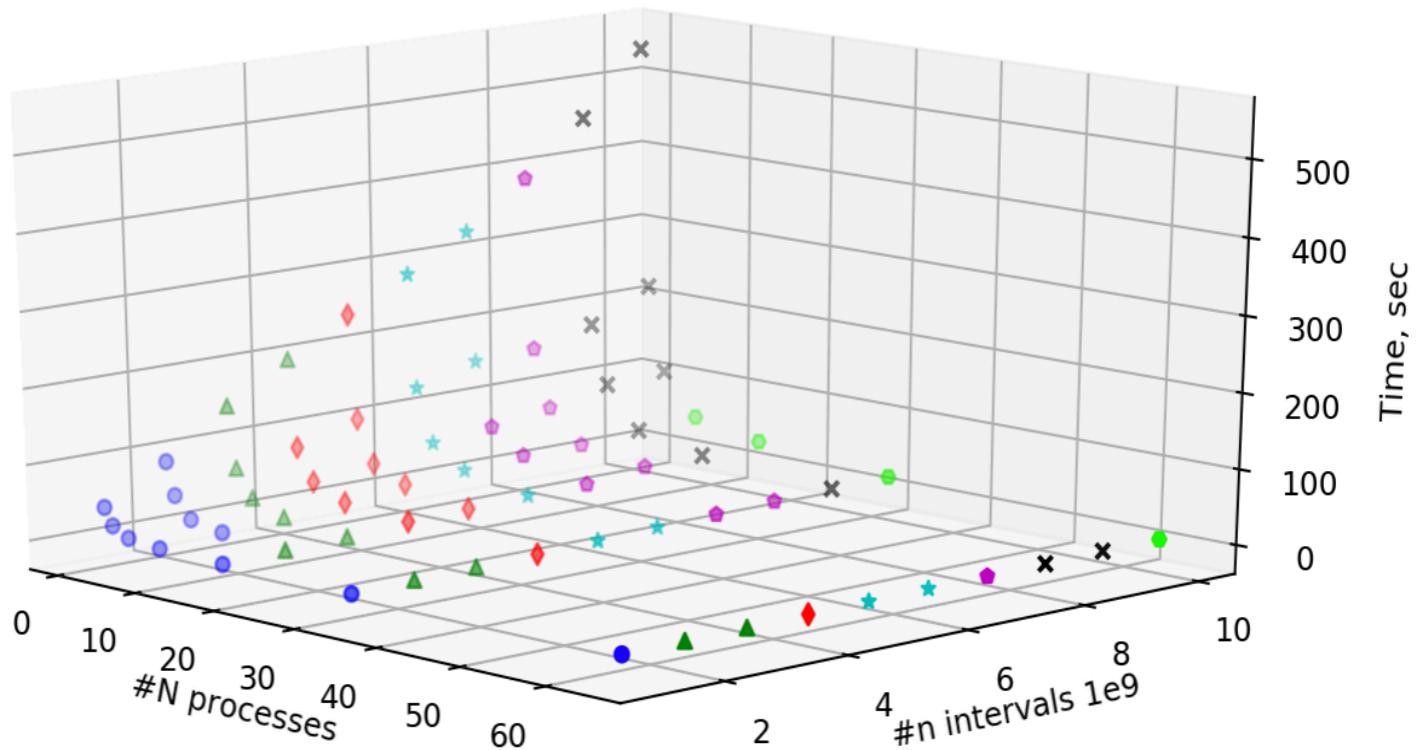
5.2.4 Только запуски с большим количеством процессов



5.2.5 Пространственная визуализация (BlueGene)



5.2.6 Пространственная визуализация (Polus)



6. Анализ результатов

Задачу удалось значительно ускорить с применением технологий OpenMP и MPI. В результате были сделаны наблюдения:

- Для OpenMP существует предел количества нитей (в эксперименте с числом интервалов $1e8 - 96$), когда время перестает значительно меняться. Связано это с увеличением накладных расходов. При использовании MPI такой ситуации не наблюдалось – время исполнения практически линейно зависит от числа процессов.
- Задача хорошо распараллеливается, т.к. нет какой-либо зависимости по данным – все нити/процессы выполняют подсчет интеграла на своих отрезках и взаимодействие происходит непосредственно при редукции результата.
- Одновременный подсчет суммы на четных и нечетных позициях дает прирост производительности, т.к. уменьшается число дополнительных операций проверки четности.
- С целью уменьшения накладных расходов была сделана попытка имплементировать реализацию функции в место её вызова, однако прироста производительности это не дало – компилятор с уровнем оптимизации (-O3) самостоятельно выполняет встраивание простых функций в место вызова.
- При сравнении суперкомпьютеров Polus и BlueGene выявлено, что первый работает быстрее при равном числе процессов. Однако, при увеличении числа процессов BlueGene уверенно побеждает. Т.е. многое менее производительных узлов работают быстрее небольшого числа более производительных ядер Polus-a.
- Можно отметить, что производительность MPI и OpenMP при числе нитей/процессов от 1 до 8 одинакова и пропорциональна этому числу, затем в OpenMP каждая следующая нить начинает давать всё меньший вклад в скорость работы, в MPI же зависимость скорости работы от числа процессов продолжает оставаться линейной.

7. Выводы

Выполнена разработка программы для подсчета значения интеграла методом Симпсона. Были реализованы последовательная и параллельная версии с использованием технологий OpenMP и MPI, проведено тестирование на суперкомпьютерах Polus и BlueGene. Исследована масштабируемость параллельной версии программы, выполнен анализ результатов, произведено сравнение используемых технологий.