Отчет

Наименование и краткая характеристика CPU:
Architecture: x86_64
Model name: Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz
Thread(s) per core: 2
Core(s) per socket: 20
Address sizes: 46 bits physical, 48 bits virtual

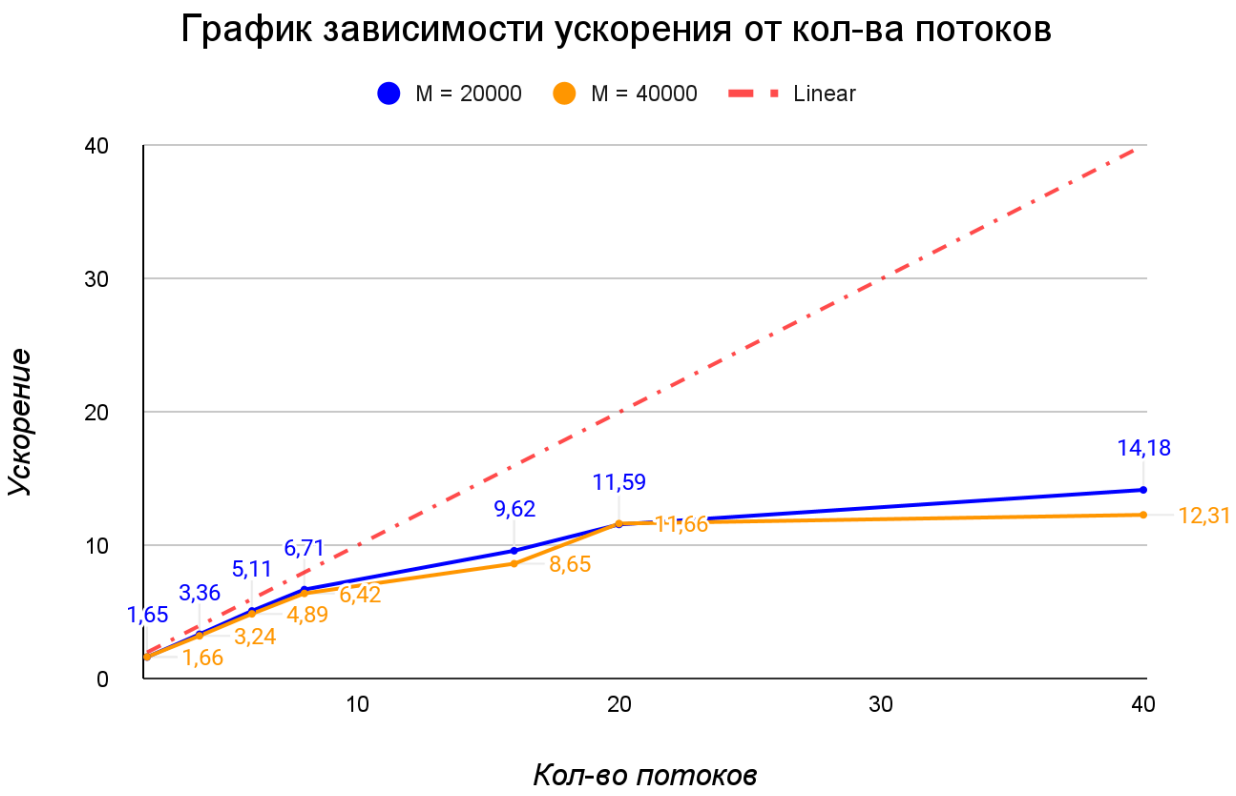
Наименование сервера:
ProLiant XL270d Gen10

Количество NUMA node:
Available: 2 nodes (0-1):
node 0 size: 385636 MB
node 1 size: 387008 MB

Операционная система:
Ubuntu 22.04.3 LTS

Задание:
Реализовать многопоточную версию программы умножения матрицы на вектор с параллельной инициализацией массивов. Провести анализ масштабируемости (для 1,2,4,7,8,16,20,40 потоков).

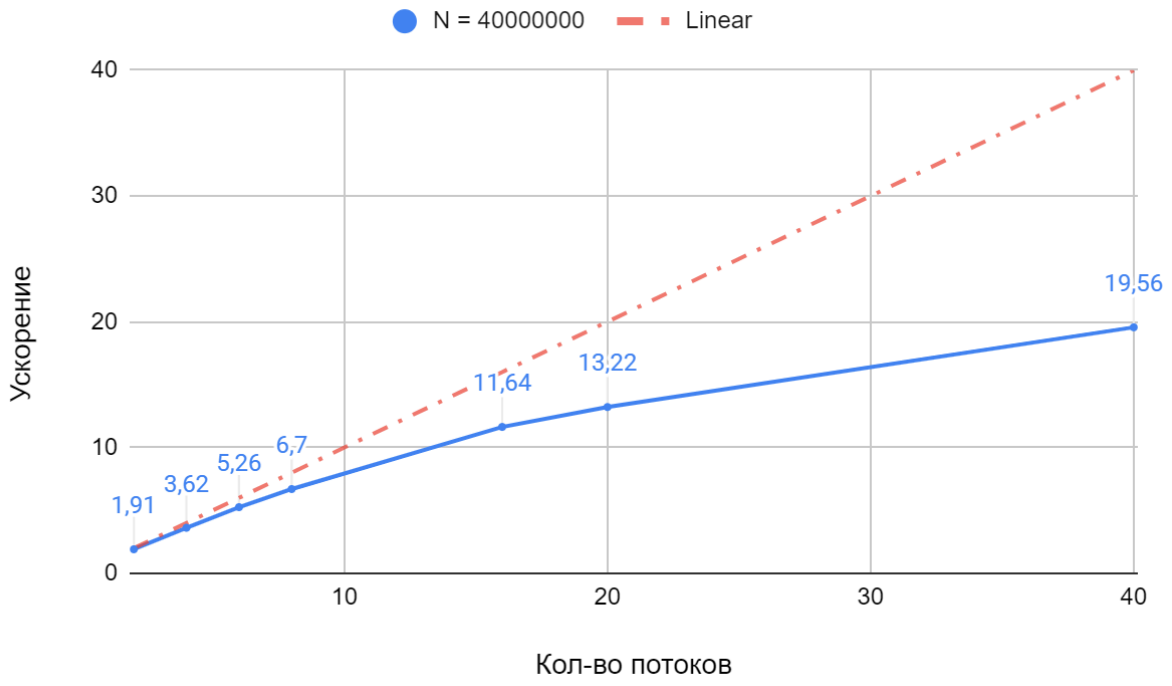
M=N	Количество потоков								
	2			**4**		**6**		**8**	
	T1 ms	*T2 ms*	*S2*	*T4 ms*	*S4*	*T6 ms*	*S6*	*T8 ms*	*S8*
20000	5332	3225	**1,65**	1588	**3,36**	1044	**5,11**	795	**6,71**
40000	21267	12815	**1,66**	6557	**3,24**	4350	**4,89**	3313	**6,42**
	16			**20**			**40**		
	T16 ms		*S16*	*T20 ms*		*S20*	*T40 ms*		*S40*
20000	554		**9,62**	460		**11,59**	376		**14,18**
40000	2460		**8,65**	1824		**11,66**	1728		**12,31**



Основываясь на графиках и таблице, можно сделать вывод, что масштабируемость при количестве потоков от 2 до 8 увеличивалась, а при использовании более 16 потоков масштабируемость становится довольно низкой.

Задание:
Реализовать параллельную версию программы численного интегрирования, написать код функции с использованием «#pragma omp atomic» и локальной переменной.

nsteps	Количество потоков								
	2			**4**		**6**		**8**	
	T1 ms	*T2 ms*	*S2*	*T4 ms*	*S4*	*T6 ms*	*S6*	*T8 ms*	*S8*
40000000	489	256	**1,91**	135	**3,62**	93	**5,26**	73	**6,70**
	16			**20**			**40,00**		
	T16 ms		*S16*	*T20 ms*		*S20*	*T40 ms*		*S40*
	42		**11,64**	37		**13,22**	25		**19,56**



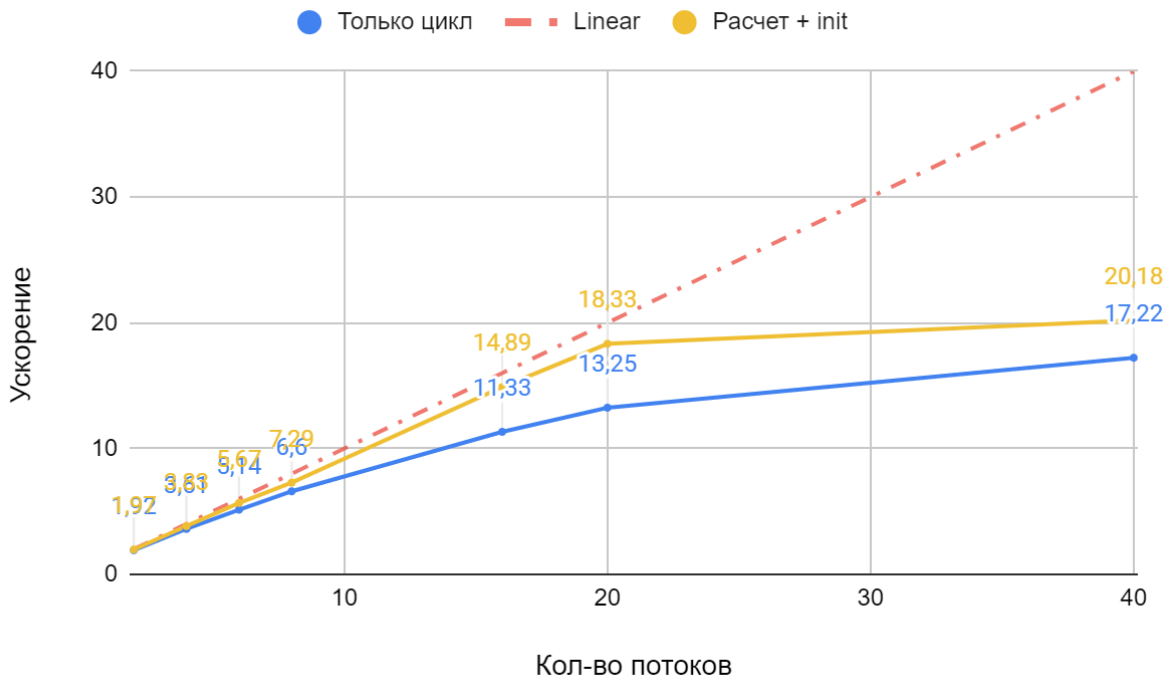
Основываясь на графике и таблице, можно сделать вывод, что до 16 потоков масштабируемость возрастает почти линейно, далее увеличивая кол-во потоков масштабируемость заметно падает, следовательно, в этой задаче не имеет смысла использовать больше 16 потоков.

Задание:
Решить СЛАУ методом простой итерации. Написать код на c++.

Исходные данные:
Элементы главной диагонали матрицы A равны 2.0, остальные равны 1.0.Все элементы вектора b равны N+1. В этом случае решением системы будет вектор, элементы которого равны 1.0. Начальные значения элементов вектора x можно взять равными 0.

arr_elem	Количество потоков (только for)								
	2			**4**		**6**		**8**	
	T1 ms	*T2 ms*	*S2*	*T4 ms*	*S4*	*T6 ms*	*S6*	*T8 ms*	*S8*
5000	10745	5605	**1,92**	2978	**3,61**	2089	**5,14**	1628	**6,60**
	16			**20**			**40,00**		
	T16 ms		*S16*	*T20 ms*		*S20*	*T40 ms*		*S40*
	948		**11,33**	811		**13,25**	624		**17,22**

arr_elem	Количество потоков (for + init)								
	2			**4**		**6**		**8**	
	T1 ms	*T2 ms*	*S2*	*T4 ms*	*S4*	*T6 ms*	*S6*	*T8 ms*	*S8*
5000	10795	5493	**1,97**	2817	**3,83**	1904	**5,67**	1481	**7,29**
	16			**20**			**40,00**		
	T16 ms		*S16*	*T20 ms*		*S20*	*T40 ms*		*S40*
	725		**14,89**	589		**18,33**	535		**20,18**



Вывод:

Использование #pragma parallel for num_threads(t) имеет смысл только при распараллеливании цикла, отвечающего за расчеты, потому что если использовать такое распараллеливание при инициализации массивов и присваивании значений, то время работы программы становится больше, особенно это заметно если запускать программу на большом кол-ве потоков.

Если выделить часть кода, которая будет выполняться параллельно и там уже провести инициализацию массивов, то можно заметить, что масштабирование до 20 потоков почти линейное, хотя если у нас менее 10 потоков, распараллеливать инициализацию не имеет смысла, т.к масштабирование практически не меняется