Chido Nguyen – 931506965 – nguychid@oregonstate.edu

Project 1: OpenMP Numeric Integration

1. Project 1 was tested on OSU flip 2 servers. (flip2.engr.oregonstate.edu). Reliability of servers would be considered good. When uptime command was ran CPU load averages were between 1.5 – 3.3. In the recent minute it was around 1.5 and for CPU load averages at the last 5 mins was about ~3.4.

2. Volume looks to be ~ **25.3125 units^3** (at 30k nodes 25.312500004 value).

3.**Tables document average performance**

Table 1.a : Shows the calculation speed for each thread count at a specific node value. Speed is in MegaCalculations per second or MC/s for short.

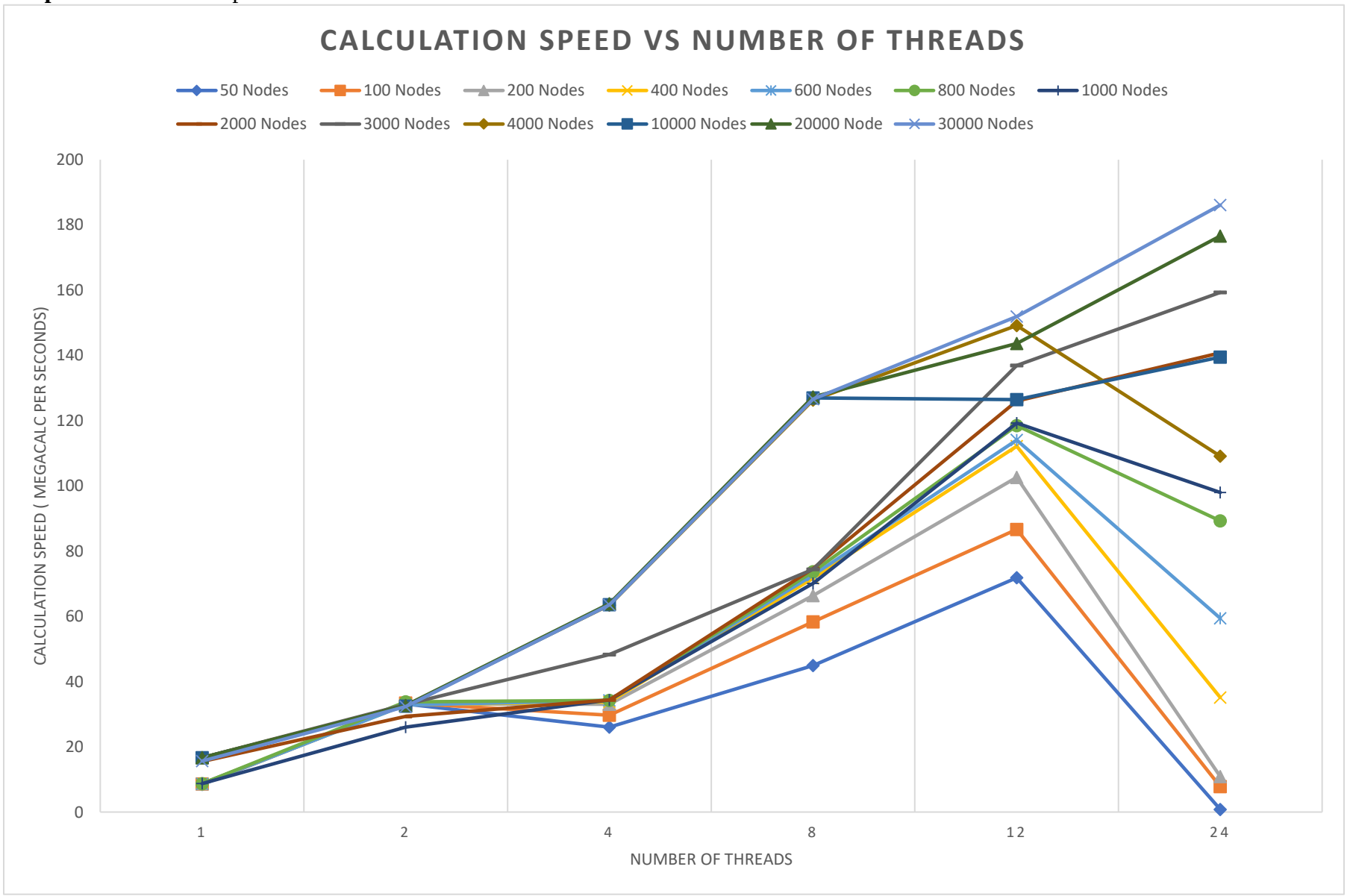
	Average	MC/s @ 10 Nodes	MC/s @ 50 Nodes	MC/s @ 100 Nodes	MC/s @ 200 Nodes	MC/s @ 400 Nodes	MC/s @ 600 Nodes	MC/s @ 800 Nodes
Thread	1	5.42	8.49	8.63	8.71	8.71	8.72	8.72
Thread	2	26.89	33.14	33.41	33.11	32.88	32.82	33.8
Thread	4	10.79	26.03	29.7	33.08	33.83	34.23	34.27
Thread	8	14.01	44.89	58.28	66.4	71.74	72.67	73.69
Thread	12	14.83	71.83	86.67	102.62	112.17	114.06	118.5
Thread	24	0.03	0.82	7.79	10.89	35.14	59.43	89.3

Table 1.b: Shows the calculation speed for each thread count at a specific node value. Speed is in MegaCalculations per second or MC/s for short.

	Average	10MC/s @ 1000 Nodes	MC/s @ 2000 Nodes	MC/s @ 3000 Nodes	MC/s @ 4000 Nodes	MC/s @ 10000 Nodes	MC/s @ 20000 Nodes	MC/s @ 30000 Nodes
Thread	1	8.73	15.5	16.68	16.67	16.66	16.66	15.64
Thread	2	26.03	29.31	32.99	32.61	32.46	32.7	32.3
Thread	4	34.37	34.41	48.29	63.45	63.57	63.88	63.48
Thread	8	70.08	74.67	74.43	126.32	126.95	127.23	126.6
Thread	12	119.27	126.03	136.92	149.18	126.46	143.67	151.95
Thread	24	97.95	140.8	159.33	109.11	139.48	176.63	186.1

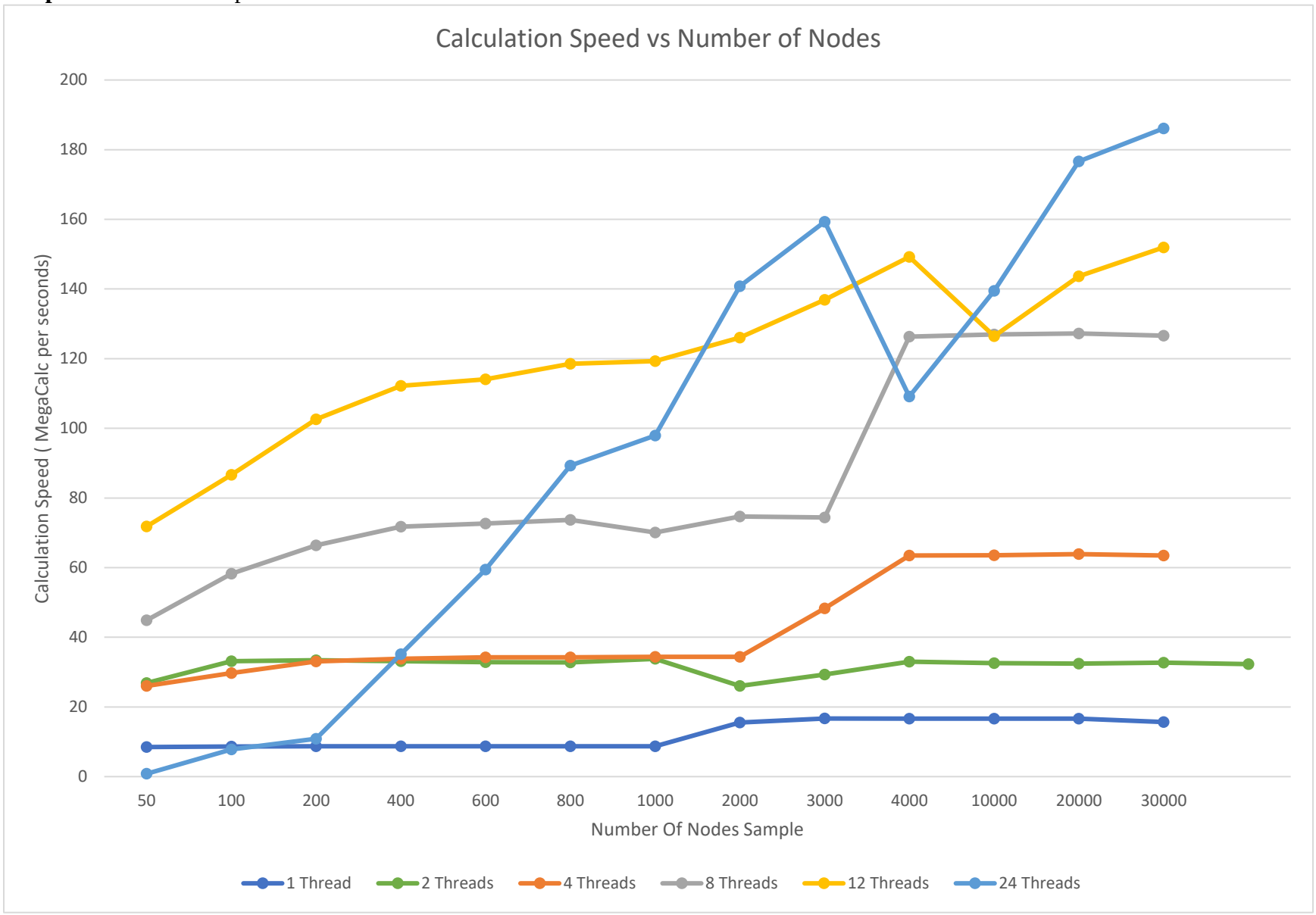
Chido Nguyen – 931506965 – nguychid@oregonstate.edu

Graph 1: Calculation Speed as Thread count increases



Chido Nguyen – 931506965 – nguychid@oregonstate.edu

Graph 2: Calculation Speed as Nodes increase



Chido Nguyen – 931506965 – nguychid@oregonstate.edu

4. Observation of patterns in the speed:

Ignoring any potential outliers; the first observation I see in the speed my program produced is that as we increased the thread count our calculations done per second was also increased as well. We can observe this either through table 1.a/1. b by going down the column which keeps a constant data sample size and changes thread count. Or by looking at either of the two graphs; graph 1 shows us that increasing our thread count generates a positive slope in our lines where as for graph 2 the higher the thread count the bigger its horizontal axis (y-value) value is.

Graph 1 shows us more clearly than graph 2 that as we increase the number of cores we should see an increase in calculation speed (MC/s) as we go across the x-axis. We observe this to a certain degree. There are some anomalies with various dips and dramatic drop offs or super slow start from higher core counts. These will be addressed in the next "Why" section.

Graph 2 when in the "norm" shows us a clearer leap in speed and what the potential speed ceiling is for our code using different thread counts. For example, thread 1 looks to level out right below 20 MegaCalc/s (from our table its around 16-17 max). Then we see that at 2 there is a clear gap in speed of calculations.

We observe that for certain out of the normal data points and lines it dips or runs slower than what we would intuitively think otherwise. This will be addressed in the next "Why" section as well.

The KEY take away in my opinion is that at the largest data sample size (30,000 nodes) we see that 24 threads > 12 > 8 > 4 >2 > 1 in speed; as shown in graph 2 with the distinct jumps in speed vertically at 30k node sample.

5. Why?

Normal Expectations:

First, we see an increase in our calculation speed up because we're increasing the thread count where work is being done. Logically speaking if we are able to do more work within a given time then our speed of getting work done increases. I.e. a cook needs to prepare a feast for a party. With one cook he or she can only prepare one dish at a time, where as now we increase the # of cooks (threads) in the kitchen to 4 we have 4 cooks preparing 4 different dishes in the same amount of time it would've taken 1 cook to do 1 dish. Nothing too special here but logical assumption that by increasing the "work engine" we can increase the work output.

Unexpected Expectations:

I first want to address the random dips in what should be an increase in slope or a positive growth rate graph. I wrote a bash script to alternate between testing for various core counts and data size documented above. I can the script and it ran through everything automatically. I noticed that even through multiple runs I would have random data points drop off. My assumption is that the server is probably getting used by other students and during certain

calculation points a different student could've been using the system and some resources were diverted towards the students causing the dips in speed.

Second to look at why 24 – thread was losing out to other threads in terms of speed as shown in our table 1 and graph 1 where it dipped down-wards. I think this comes down to the cost of thread over-head outweighed the its benefit at lower data samples. We see that at anything above 2000 node samples the lines representing node sizes start going positive in graph 1. As opposed to being negative from 1-1000 node samples. With an exception of node sample size at 4000 where I believe the server might've been used by a separate student causing it to slow down a bit, but I could be wrong with these assumptions. The key idea here is that with more thread we'd want a big enough dataset to make the increase in thread-overhead worthwhile rather than having it provide diminishing returns. Look at our table 1.a 1-thread ran 10 nodes at 5.4 MC/s where as 24 threads ran at .03 MC/s.

**I attempted to run my script multiple times and never once was able to get a smooth increase performance, there were always 1 or more outlier data samples where it was lower than what was expected. Rather than running just that scenario separately, I chose to keep the data to maintain the integrity of the rest of the other data points. **

Chido Nguyen – 931506965 – nguychid@oregonstate.edu

6. Parallel Fraction + 7 Max Speed up:

Fp = (n)/ (n-1) *(T1-Tn) / T1

n = 1 or 2 or 4 or 8 or 12 or 24

Tn = (Number of Nodes) ^ 2 / ((Calculation Speed at 30k node samples) * 1,000,000)

		Execution Time (sec)	**Parallel Fraction (Fp)**
Thread	1	57.544	**Reference**
Thread	2	27.864	**1.03157**
Thread	4	14.177	**1.00482**
Thread	8	7.109	**1.00165**
Thread	12	5.923	**0.978604**
Thread	24	4.836	**0.955767**

Max speed up is lim as x approaches infinity SpeedUp = 1 / ((Fp/n) + (1-Fp));

As n gets closer to infinity we can say that Fp/n = 0, leaving us with
Max Speed = 1/(1 – Fp) ➔ 1 / (1 - .955767) ➔ **22.60756**