

עבודת הגשה 2 – מערכות הפעלה:

Q1. Amdahl's Law:

$$S = (1-P) + P/N$$

1. **Given:** 75% of the execution time is parallelizable, and there are 4 processors.

$$\frac{1}{0.25 + \frac{0.75}{4}} = \frac{16}{7} = 2.2857$$

2. **Given:** 60% of the program is parallelizable, and we need a speedup of 3.

$$\frac{1}{0.4 + \frac{0.6}{n}} = 3$$

$$1 = 1.2 + 1.8n$$

$$N = -9$$

Since this leads to a negative result which is not possible, it implies that the desired speedup of 3 cannot be achieved with a program where only 60% can be parallelized.

3. **Given:** 90% of the execution time is parallelizable, and there are 32 processors.

$$\frac{1}{0.1 + \frac{0.9}{32}} = \frac{320}{41} = 7.8048$$

4. **Given:** 85% can be parallelized, and we have an infinite number of processors.

As N approaches infinity, P/N approaches 0

$$\frac{1}{0.15 + \frac{0.85}{\infty}} = \frac{20}{3} = 6.667$$

5. **Given:** 80% of the code is parallelizable, and we need to compare the speedup with 4 and 32 processors.

$$\frac{1}{0.2 + \frac{0.8}{4}} = \frac{5}{2} = 2.5$$

$$\frac{1}{0.2 + \frac{0.8}{32}} = \frac{40}{9} = 4.444$$

The speedup changes from 2.5 to 4.444 when increasing the number of processors from 4 to 32.

Q2. Process vs. Thread

1.

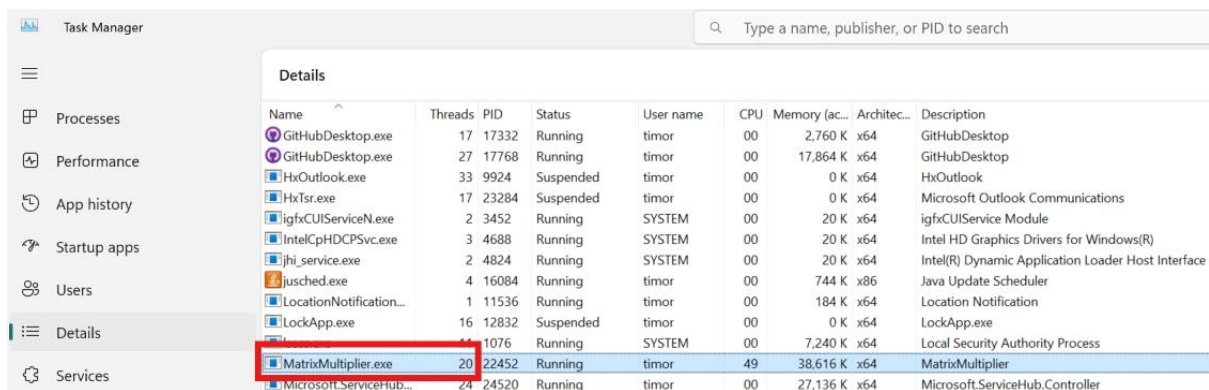
Iteration	Process Time (ms)	Thread Time (ms)
1	18.3320	4.1388
2	19.5788	4.3289
3	21.2743	4.3849
4	19.0686	5.2090
5	29.0086	4.3778
6	20.1322	5.2824
7	20.0872	8.7178
8	70.1976	14.1264
9	30.9396	4.9438
10	21.3926	4.3189

2. The measured times demonstrate that creating and executing threads is significantly faster than processes. This is due to the shared memory space and lower overhead associated with thread creation and management compared to processes, which require more substantial setup and memory allocation.

Switching between threads is faster because they share the same process context. In contrast, processes require a complete context switch, which involves saving and loading memory maps, file descriptors, and other process-specific information.

Adding printouts with random numbers helps to ensure that each run is performing some action, which prevents the compiler or runtime from optimizing away the work entirely. It also introduces a slight variation in the execution time, which can affect the I/O performance slightly, providing a more realistic measurement of time required to manage processes and threads.

Q3. Multithreaded Matrix Multiplication



Name	Threads	PID	Status	User name	CPU	Memory (ac...	Archite...	Description
GitHubDesktop.exe	17	17332	Running	timor	00	2,760 K	x64	GitHubDesktop
GitHubDesktop.exe	27	17768	Running	timor	00	17,864 K	x64	GitHubDesktop
HxOutlook.exe	33	9924	Suspended	timor	00	0 K	x64	HxOutlook
HxTsr.exe	17	23284	Suspended	timor	00	0 K	x64	Microsoft Outlook Communications
igfxCUIServiceN.exe	2	3452	Running	SYSTEM	00	20 K	x64	igfxCUIService Module
IntelCpHDCPSvc.exe	3	4688	Running	SYSTEM	00	20 K	x64	Intel HD Graphics Drivers for Windows(R)
jhi_service.exe	2	4824	Running	SYSTEM	00	20 K	x64	Intel(R) Dynamic Application Loader Host Interface
jusched.exe	4	16084	Running	timor	00	744 K	x86	Java Update Scheduler
LocationNotification...	1	11536	Running	timor	00	184 K	x64	Location Notification
LockApp.exe	16	12832	Suspended	timor	00	0 K	x64	LockApp.exe
Local Security Authority Process	14	1076	Running	SYSTEM	00	7,240 K	x64	Local Security Authority Process
MatrixMultiplier.exe	20	22452	Running	timor	49	38,616 K	x64	MatrixMultiplier
Microsoft.ServiceHub...	24	24520	Running	timor	00	27,136 K	x64	Microsoft.ServiceHub.Controller

Q4 .Multithreaded MergeSort

a) The multi-threaded merge-sort algorithm extends the traditional merge-sort by using multiple threads to perform sorting tasks concurrently. The strategy involves dividing the input array into smaller sub-arrays, sorting each sub-array concurrently using separate threads, and then merging the sorted sub-arrays. The main steps are as follows:

1. **Divide:** The array is recursively split into two halves until the size of each sub-array is less than or equal to a specified threshold (n_{Min}). At this point, the sub-array is small enough to be sorted directly using a single thread.
2. **Conquer:** Each half is sorted recursively in parallel using separate threads. This is achieved using the fork/join strategy where each recursive call to sort a sub-array is forked into a new thread.
3. **Combine:** The sorted halves are merged back together to form a single sorted array. The merging process involves comparing elements from each half and combining them into a single sorted array.

b) multi-threaded merge sort diagram:

