CSCE 313-200: Computer Systems

Homework #2 (100 pts)

Due date: 3/18/25

1. Purpose

Experiment with advanced synchronization primitives in Windows and understand how to design high-performance algorithms for parallel execution.

2. Problem Description

This project builds on the concepts developed in homework #1. The goal of this homework is to crawl large planets of CC 2.x using BFS (the other three methods are not needed) and build a distribution for the number of nodes found at each distance from the rover. While finding the exit is still important, the central algorithmic challenge here is how to guarantee correct BFS search under non-deterministic conditions of multi-threading. Observe that homework #1's BFS was only an *approximation* to the true BFS assumed in graph theory (i.e., nodes at distance *i* are explored only when all nodes at distance *i*–1 have been visited). In the previous formulation, nothing prevented threads from operating on nodes at arbitrary distances *i* and *j* at the same time.[1] If left uncorrected, this may lead to a false conclusion about the true shortest distance to the exit.

The goal of this homework is to implement a *provably correct* parallel BFS without making any assumptions on the OS scheduler and experiment with it in reasonably large caves (up to 4 billion rooms). For this to be feasible, many parts of your code must be optimized and tuned to run at maximum speed. While homework #1 utilized less than 10% of the total CPU capacity, this version should be able to scale to all available cores and keep them close to 100% utilization.

2.1. Code (75 pts)

The rules related to poor coding practices from section 2.1 of homework #1 apply here; **however, all constructs/functions/classes from C++ libraries now disallowed**. You are limited to *two* shared BFS queues and *one* shared hash table (you can use additional queues local to each thread as needed, as well as shared variables related to statistics) . Unlike the previous homework, you will also be graded on speed that is based on code efficiency rather than pipe-communication delay, with an expectation of rates close to 10M rooms per second (rps) on class server ts.cse.tamu.edu or quad-core Azure servers (instances called B4ms).

There are several printouts that are mandatory. First, the stats thread must print in the following format every 2 seconds:

[1] Suppose $thread_k$ pulls from the Q one of the nodes at distance 1 and then explores it very slowly (i.e., gets blocked by the kernel for an hour). In the meantime, the remaining threads may crawl every possible level *j* and finish exploration of the whole cave.

```
[428.8M] U 518.9M D 947.7M, 9.38M/sec, 1497*, 7% uniq [99% CPU, 3950 MB]
```

where this example shows that 428.8M nodes have been removed from the BFS Q and sent for exploration, 518.9M rooms are still left to be explored, 947.7M unique rooms have been discovered thus far, 9.38M/sec is the current rate at which rooms are extracted from the Q, 1497 threads are active (i.e., working on exploring some rooms), 7% of the rooms returned from the robots were unique, 99% of the CPU is currently utilized, and the program is using 3.95 GB of RAM. The rps rate, percent unique, and CPU utilization are averaged over the last 2 seconds. The number of active threads is sampled immediately before the printout is made. The other parameters are cumulative.

Once the exit is found, the program should print this fact and *continue running* until the whole cave has been covered:

```
Thread [1341]: found exit room 1C63A9F, distance 12, rooms explored 555,685,089
```

The program must make periodic announcements about the number of nodes at each distance (also called depth/level) as it learns their values:

```
--------- Switching to level 1 with 6 nodes
--------- Switching to level 2 with 33 nodes
--------- Switching to level 3 with 162 nodes
--------- Switching to level 4 with 745 nodes
--------- Switching to level 5 with 3,269 nodes
--------- Switching to level 6 with 14,738 nodes
--------- Switching to level 7 with 196,291 nodes
--------- Switching to level 8 with 1,592,297 nodes
--------- Switching to level 9 with 12,294,620 nodes
--------- Switching to level 10 with 87,884,288 nodes
--------- Switching to level 11 with 421,068,639 nodes
--------- Switching to level 12 with 471,263,881 nodes
--------- Switching to level 13 with 73,733,995 nodes
--------- Switching to level 14 with 5,310,989 nodes
--------- Switching to level 15 with 352,919 nodes
--------- Switching to level 16 with 23,308 nodes
--------- Switching to level 17 with 1,531 nodes
--------- Switching to level 18 with 107 nodes
--------- Switching to level 19 with 5 nodes
```

Note that comma separation in these numbers is also required. As there are no library functions that can print in this format, you will have to write your own. Once all levels are done, you must properly shutdown CC 2.x and print the following:

```
Execution time: 117.59 sec
Average speed: 9.13M/sec
```

If your code is properly optimized, the average speed should be close to 8M/sec on our server, with the peak rate hitting 14M/sec.

2.2. Report Requirements (25 pts)

Several things to analyze:

1. Run your homework #1 and find an example where its BFS produces an incorrect distance to the exit (i.e., the reported distance is not the shortest). Trace how it happened and explain in detail the underlying cause.

2. Show a benchmark illustrating how fast the BFS design of homework #1 would handle discovered rooms in CC 2.x if it were augmented with batching, but the remaining algorithms stayed the same. Convert hw#1 to use 10K batch size with CC 2.x and run a test on a sufficiently large planet.
3. Experiment with STL queue/set and build a model for their overhead per DWORD element inserted into them. To accomplish this, create two objects:

```
queue<DWORD> q;
set<DWORD> m;
```

   and analyze them separately using the following procedure. First, insert N_1 = 400M elements into a given structure and record the number of bytes B_1 used by your process (e.g., as reported by TaskManager). Repeat with $N_2 = 2N_1$ items and record the corresponding B_2. Then, use these four numbers to estimate the STL overhead per element. Ballpark the amount of RAM needed to search a cave with 4 billion rooms assuming maximum occupancy of both structures.
4. Explain the design of your new algorithm and document its performance. If you went through multiple intermediate versions, list these approaches and explain why they were lacking in terms of performance and/or RAM scalability.
5. Examine how your exploration speed scales with the number of cores. Set thread affinity to limit your code to 1, 2, …, K CPUs and plot the corresponding search rate vs the number of allowed cores. Examine how well this curve approximates a linear function.

2.3. Extra Credit

If your program is able to finish P30, cave 55 with 1500 threads and keep peak RAM usage below 5 GB, you will receive 10 extra points. Additionally, if you can noticeably improve the runtime by avoiding synchoronization when the fraction of unique rooms gets close to zero, you will get another 10 points. The target delay on P30, cave 55 using ts.cse.tamu.edu or Microsoft Azure's quad-core B4ms is around 130 seconds (1500 threads).

3. Details

3.1. CC Protocol

Exchange of messages with the CC is identical to that in homework #1. The only exception is that now planet *P* contains 2^P rooms in each cave and you are limited to planets 20-32 (i.e., 1M – 4B rooms).

3.2. Robot Protocol

Pipe names and robot operation (i.e., commands MOVE, CONNECT, DISCONNECT) are the same, but the format of all messages has been simplified and compressed to carry useful information in fewer bytes. Robot commands contain a fixed header (shown below) followed with an optional list of requested rooms. This list is only applicable when `command == MOVE` and should not be present otherwise. The maximum number of rooms in a batch is 10K and there is no need to explicitly specify the length of this list since the

robot automatically computes the number of rooms based on the number of bytes that come out of the pipe. Unlike the previous homework, each room ID is a DWORD and the header is given by:

```
class CommandRobotHeader {
public:
        DWORD           command;
};
```

To make CC.exe faster and leaner, its algorithm was changed to avoid explicitly building the graph. Instead, it is stored implicitly as a randomized set of rules for generating neighbor lists. To keep overhead minimal, caves in CC 2.x do not have light intensities and thus cannot be searched using bFS/A*. While there are no monsters and all responses are legitimate, you still need to check for possible API errors, report abnormal conditions to the screen, and terminate your program upon detecting failure.

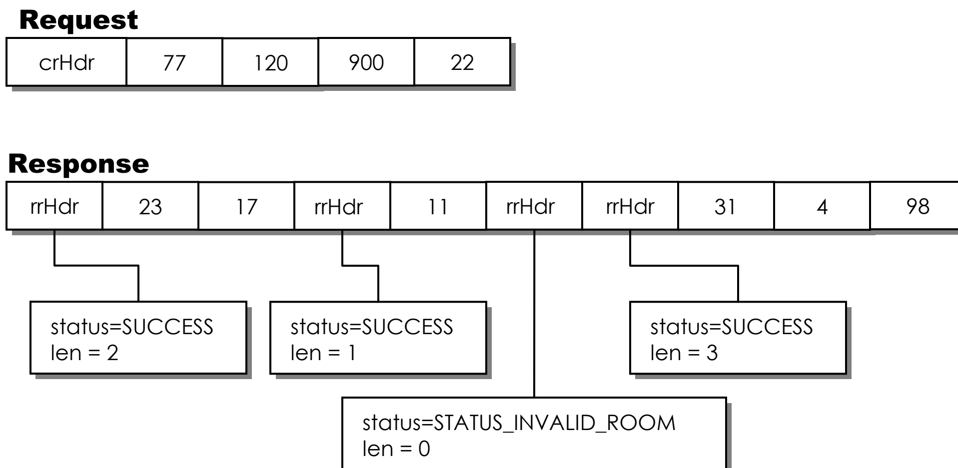

Figure 1. Example with four requested rooms.

Suppose a particular request carries *N* room IDs. Then, the response will be a single message consisting of *N segments*, where each segment contains a response header and a list of neighboring room IDs. The header is a DWORD in which the first (i.e., lower) 3 bits specify the status of the command and the remaining 29 bits specify the length of the list that follows. If an error occurs, the status field identifies the type of error. For the example in Figure 1, the initial request carries four rooms – 77, 120, 900, and 22. The corresponding response provides two neighbors for room 77, one for 120, three for 22, and an error for 900. To avoid working with bit shifts, the following class is useful:

```
class ResponseRobotHeader {
public:
        DWORD           status:3;                       // up to 8 unique result codes
        DWORD           len:29;                         // max len = 0.5B neighbors
};

#define STATUS_OK                       0               // no error, command successful
#define STATUS_ALREADY_CONNECTED        1               // repeated attempt to connect
#define STATUS_INVALID_COMMAND          2               // command too short or invalid
#define STATUS_INVALID_ROOM             3               // room ID doesn't exist
#define STATUS_INVALID_BATCH_SIZE       4               // batch size too large or equals 0
#define STATUS_MUST_CONNECT             5               // first command must be CONNECT
```

3.3. Parallel BFS

To implement a correct multi-threaded BFS, it is sufficient to prevent rooms at depth d from being mixed with rooms at depth $j \geq d + 1$ in the same queue. In fact, it is possible to enforce strict BFS order by maintaining just two queues Q_c and Q_f. The former keeps unexplored nodes that belong to the *current* depth d, while the latter stores *future* rooms at depth $d + 1$ as they are being discovered. As a result, BFS always extracts from Q_c and writes newly found rooms to Q_f.

Once Q_c becomes empty and no active threads are left, BFS upgrades to level $d + 1$, moves all elements from Q_f to Q_c, and continues as described in the preceding paragraph. In practice, however, no movement of elements between Q_f and Q_c takes place. Instead, one declares an array of two queues arrQ[2] and simply alternates the current index between 0 and 1, i.e., arrQ[cur] is always Q_c and arrQ[cur^1] is always Q_f. *Do not physically copy one queue into the other!*

3.4. Performance

The goal here is to speed up the code through four main techniques: faster synchronization, batching of pipe requests, faster verification of uniqueness for discovered nodes, and more RAM-efficient BFS queues.

The first issue, i.e., synchronization, has been extensively discussed in class. Using the performance of various APIs shown in slides and considering the requirement for batch-mode push/pop, select the most appropriate producer-consumer algorithm and deploy it in this homework. The second issue is already supported by CC 2.x and requires small tweaks to the BFS loop to enable batching over pipes.

The third improvement comes from replacing an STL set with a hash table. Keep in mind that you cannot store all discovered nodes in RAM (because 4B rooms require 16GB and you need additional space for the two BFS queues), which means that you need to come up with an alternative design leveraging the fact that the number space of room IDs is limited to only 4B unique items. Under these conditions, a much more efficient representation of hash tables is possible. Once you have this working, pay special attention to how your updates to the hash table are synchronized since this will be your major bottleneck. *For the extra credit, attempt to design a system that synchronizes as little as possible*.

Finally, you will need to write your own queues using the following skeleton:

```
class MyQueue {
	HANDLE heap;			// custom heap
	DWORD  *buf;			// buffer pointer
	DWORD  head, tail;		// usual head/tail offsets
	DWORD  spaceAllocated;		// buffer size
	DWORD  sizeQ;			// number of items in Q
public:
	void Push (DWORD item);			// single push
	DWORD Pop (DWORD *array, int batchSize);	// batch pop
};
```

3.5. Caveats

Recall that holding any critical section for too long is costly. For a MOVE with 10K rooms, the response may come back with 40-80K neighbors. Thus, instead of locking the mutex for the entire cycle of uniqueness verification, a much better approach is to use non-mutexed (i.e., interlocked) access to the hash table and append unique nodes to some *local* queue within each thread. Later, once you know all unique rooms, they can be off-loaded into Q_f inside a short critical section. In order to avoid synchronizing threads on access to the main process heap while working with local queues, it is best to run each queue with its own copy of the heap. See below for the APIs.

Note that you should avoid looping on each element while doing batch-mode pop. Instead, use memcpy to shovel large contiguous chunks from the internal buffer back into the caller-provided array. Parameter batchSize in the argument to `MyQueue::Pop` specifies the maximum number of items that the caller's array can accept. The function extracts up to that many elements and provides the actual number in the return value.

Note that `MyQueue::Push` should dynamically expand the buffer to accommodate new items. You can use the rules of STL, i.e., double the array each time an item cannot fit into the existing buffer.

3.6. APIs

Creation of a heap that bypasses internal mutexes is accomplished using the following:

```
HANDLE heap = HeapCreate (HEAP_NO_SERIALIZE, ..., 0);
char *ptr = (char*) HeapAlloc (heap, HEAP_NO_SERIALIZE, ...);
HeapFree (heap, HEAP_NO_SERIALIZE, ...);
HeapDestroy (heap);
```

Note that without synchronization such heaps *can only be used within a single thread*. However, if you mutex around every HeapAlloc/HeapFree operation, then shared queues relying on non-serialized heaps can work as well.

The course website has a sample project that includes the CPU class (cpu.cpp and cpu.h), which can be used to obtain CPU utilization and RAM usage needed for the stats.

Since you will be approaching 100% CPU utilization, make sure to set all your worker threads to idle priority. You should also monitor memory usage in Task Manager to make sure it does not exceed the available RAM during runs on large planets; otherwise, the program will become super slow and the system may hang for long periods of time.

4. Traces

The trace below was obtained on a 12-core AMD server ts.cse.tamu.edu.

```
*** CC v2.1: starting with PID 5376 (hex 1500)
*** CC: found 12 CPUs, 14130.12 MB of free RAM
Opened (planet 30, cave 55) with 1500 robot(s)

--------- Switching to level 1 with 6 nodes
--------- Switching to level 2 with 33 nodes
--------- Switching to level 3 with 162 nodes
--------- Switching to level 4 with 745 nodes
--------- Switching to level 5 with 3,269 nodes
```

```
  [0.0M] U 0.0M D 0.0M, 0.00M/sec, 1*, 100% uniq [5% CPU, 593 MB]

--------- Switching to level 6 with 14,738 nodes
--------- Switching to level 7 with 196,291 nodes
--------- Switching to level 8 with 1,592,297 nodes

  [1.8M] U 0.2M D 2.1M, 0.90M/sec, 156*, 100% uniq [8% CPU, 615 MB]

--------- Switching to level 9 with 12,294,620 nodes

  [14.1M] U 5.2M D 19.3M, 6.15M/sec, 1151*, 98% uniq [81% CPU, 1053 MB]
  [14.1M] U 63.9M D 78.0M, 0.00M/sec, 352*, 94% uniq [71% CPU, 1133 MB]

--------- Switching to level 10 with 87,884,288 nodes

  [30.8M] U 81.4M D 112.2M, 8.35M/sec, 1500*, 89% uniq [87% CPU, 1506 MB]
  [35.4M] U 107.4M D 142.8M, 2.31M/sec, 1500*, 83% uniq [97% CPU, 1845 MB]
  [41.9M] U 141.8M D 183.7M, 3.21M/sec, 1500*, 81% uniq [83% CPU, 2041 MB]
  [50.6M] U 184.1M D 234.7M, 4.38M/sec, 1500*, 76% uniq [80% CPU, 2070 MB]
  [59.1M] U 221.0M D 280.1M, 4.25M/sec, 1500*, 70% uniq [71% CPU, 2179 MB]
  [71.3M] U 270.5M D 341.8M, 6.11M/sec, 1500*, 65% uniq [90% CPU, 2265 MB]
  [85.5M] U 319.1M D 404.6M, 7.10M/sec, 1500*, 58% uniq [92% CPU, 2421 MB]
  [91.4M] U 337.7M D 429.1M, 2.92M/sec, 1500*, 54% uniq [89% CPU, 3281 MB]
  [102.0M] U 371.8M D 473.8M, 5.30M/sec, 1399*, 50% uniq [60% CPU, 2650 MB]

--------- Switching to level 11 with 421,068,639 nodes

  [119.6M] U 411.9M D 531.4M, 8.79M/sec, 1500*, 44% uniq [85% CPU, 2331 MB]
  [127.6M] U 428.7M D 556.4M, 4.04M/sec, 1500*, 40% uniq [97% CPU, 2631 MB]
  [141.3M] U 454.5M D 595.7M, 6.81M/sec, 1499*, 37% uniq [88% CPU, 2777 MB]
  [159.1M] U 481.3M D 640.4M, 8.90M/sec, 1500*, 33% uniq [98% CPU, 2900 MB]
  [172.5M] U 499.3M D 671.8M, 6.64M/sec, 1500*, 30% uniq [75% CPU, 2968 MB]
  [191.0M] U 518.2M D 709.3M, 9.26M/sec, 1500*, 26% uniq [96% CPU, 3119 MB]
  [212.5M] U 535.8M D 748.3M, 10.72M/sec, 1500*, 23% uniq [96% CPU, 3222 MB]
  [236.9M] U 549.3M D 786.2M, 12.09M/sec, 1500*, 20% uniq [98% CPU, 3368 MB]
  [246.1M] U 552.9M D 799.0M, 4.59M/sec, 1500*, 18% uniq [70% CPU, 3399 MB]
  [270.7M] U 558.8M D 829.5M, 12.24M/sec, 1500*, 16% uniq [96% CPU, 3586 MB]
  [296.1M] U 560.7M D 856.8M, 12.70M/sec, 1500*, 14% uniq [98% CPU, 3725 MB]
  [320.3M] U 558.9M D 879.3M, 12.10M/sec, 1500*, 12% uniq [98% CPU, 3814 MB]
  [345.8M] U 554.1M D 899.8M, 12.72M/sec, 1500*, 11% uniq [99% CPU, 3875 MB]
  [371.3M] U 546.4M D 917.7M, 12.70M/sec, 1500*, 9% uniq [98% CPU, 3975 MB]
  [397.6M] U 536.5M D 934.1M, 13.07M/sec, 1499*, 8% uniq [99% CPU, 4017 MB]
  [423.8M] U 524.4M D 948.3M, 13.13M/sec, 1500*, 7% uniq [98% CPU, 4083 MB]
  [451.1M] U 510.3M D 961.4M, 13.63M/sec, 1500*, 6% uniq [99% CPU, 4127 MB]
  [475.9M] U 496.1M D 972.0M, 12.32M/sec, 1499*, 6% uniq [97% CPU, 4172 MB]
  [505.0M] U 478.2M D 983.2M, 14.59M/sec, 1500*, 5% uniq [98% CPU, 4184 MB]
  [523.1M] U 469.6M D 992.7M, 9.01M/sec, 520*, 4% uniq [95% CPU, 4196 MB]

--------- Switching to level 12 with 471,263,881 nodes

  [549.9M] U 448.0M D 997.9M, 13.44M/sec, 1500*, 4% uniq [86% CPU, 2563 MB]
  [577.3M] U 428.3M D 1005.6M, 13.54M/sec, 1499*, 4% uniq [100% CPU, 2623 MB]
  [602.8M] U 409.3M D 1012.1M, 12.75M/sec, 1499*, 3% uniq [99% CPU, 2654 MB]

*** Thread [1080]: found exit room 1C63A9F, distance 12, steps 619,225,089

  [631.7M] U 387.1M D 1018.8M, 14.44M/sec, 1498*, 3% uniq [99% CPU, 2696 MB]
  [660.6M] U 364.2M D 1024.8M, 14.45M/sec, 1500*, 3% uniq [99% CPU, 2731 MB]
  [687.3M] U 342.5M D 1029.9M, 13.35M/sec, 1495*, 2% uniq [97% CPU, 2739 MB]
  [714.0M] U 320.5M D 1034.5M, 13.31M/sec, 1500*, 2% uniq [97% CPU, 2783 MB]
  [742.7M] U 296.4M D 1039.0M, 14.32M/sec, 1500*, 2% uniq [98% CPU, 2790 MB]
  [770.6M] U 272.6M D 1043.1M, 13.92M/sec, 1500*, 2% uniq [98% CPU, 2823 MB]
  [799.4M] U 247.6M D 1047.0M, 14.42M/sec, 1500*, 2% uniq [100% CPU, 2810 MB]
  [825.7M] U 224.6M D 1050.3M, 13.13M/sec, 1500*, 2% uniq [98% CPU, 2827 MB]
  [853.8M] U 199.8M D 1053.6M, 14.05M/sec, 1496*, 2% uniq [99% CPU, 2834 MB]
  [883.6M] U 173.2M D 1056.8M, 14.37M/sec, 1500*, 1% uniq [99% CPU, 2847 MB]
  [910.6M] U 148.9M D 1059.5M, 13.48M/sec, 1500*, 1% uniq [98% CPU, 2860 MB]
  [933.8M] U 127.9M D 1061.7M, 11.51M/sec, 1498*, 1% uniq [97% CPU, 2862 MB]
  [966.4M] U 98.2M D 1064.6M, 16.24M/sec, 1500*, 1% uniq [98% CPU, 2863 MB]
  [992.9M] U 73.9M D 1066.8M, 13.22M/sec, 1499*, 1% uniq [98% CPU, 2878 MB]
```

```
--------- Switching to level 13 with 73,733,995 nodes

  [1009.7M] U 58.4M D 1068.1M, 8.43M/sec, 1500*, 1% uniq [71% CPU, 995 MB]
  [1032.8M] U 37.0M D 1069.8M, 11.50M/sec, 1499*, 1% uniq [95% CPU, 1043 MB]
  [1059.5M] U 12.2M D 1071.8M, 13.36M/sec, 1499*, 1% uniq [98% CPU, 1076 MB]

--------- Switching to level 14 with 5,310,989 nodes

  [1073.4M] U 0.0M D 1073.4M, 6.91M/sec, 469*, 1% uniq [78% CPU, 744 MB]

--------- Switching to level 15 with 352,919 nodes

  [1073.7M] U 0.0M D 1073.7M, 0.18M/sec, 1*, 1% uniq [35% CPU, 714 MB]

--------- Switching to level 16 with 23,308 nodes
--------- Switching to level 17 with 1,531 nodes
--------- Switching to level 18 with 107 nodes
--------- Switching to level 19 with 5 nodes

  [1073.7M] U 0.0M D 1073.7M, 0.01M/sec, 1*, 1% uniq [1% CPU, 711 MB]
  [1073.7M] U 0.0M D 1073.7M, 0.00M/sec, 0*, 0% uniq [6% CPU, 522 MB]

Waiting for CC to quit...
*** CC: main thread waiting for robots...
*** CC: all robots finished
*** CC: quitting, kernel time 2.29 sec, user time 15.42 sec
Execution time: 115.89 sec
Speed 9.26M/sec
```

313 Homework 2 Code

Name: ______________________________

	Points	Break down	Item	Points
Basic Code Structure	30	5	Use of two BFS queues	
		5	PC3.4 structure	
		5	Hash table and interlocked bit test	
		5	MyQueue with batch push/pop	
		5	Stats thread	
		5	Batch operation on pipes	
Searching Planet 30	25	5	Slow speed (> 268 sec on P30 on ts)	
		5	Too much memory (> 8GB on P30)	
		5	Incorrect # of nodes for some levels	
		5	Incorrect exit room or not printed	
		5	Clean termination	
Stats Printed	10	2	Incorrect/absent unique %	
		5	Incorrect/absent CPU & RAM usage	
		3	Incorrect/absent speed	
Other	10		Numbers not comma-separated, crashing, deadlocking, etc.	
Extra credit (ts)	20	10	< 130 sec on P30, cave 55	
		10	< 5 GB on P30, cave 55, 1500 robots	

Total points: _________________

313 Homework 2 Report

Points	Item	Points
5	Show how BFS in hw1 produces incorrect distance	
5	Show how fast your hw1 would be with CC 2.x batching	
5	Compute the RAM overhead of STL queues and maps	
5	Explain the design of your algorithm and document the performance of various designs	
5	Show how the exploration speed scales with the number of cores	

Total points: ___________________