15-418 Final Project Report

Samantha Lavelle

slavelle

A Parallel, Recursive Numbrix Solver

Summary

I implemented and parallelized an algorithm to solve Numbrix puzzles of various sizes and difficulties. The parallelization is implemented in OpenMP on the GPU of the GHC machines, as well as PSC. Accuracy is maintained across various puzzle difficulty levels and sizes up to a 15x15 board, which the program is able to print for the user to inspect upon completion of the solving algorithm.

Background

I parallelized an algorithm that solves a Numbrix puzzle. Numbrix is a number puzzle in which the goal is to create a contiguous path of numbers on a (typically square) board. Some numbers are already filled in, as in a Sudoku puzzle, so the solver must work with the placement of these numbers when creating the path. In the path, the next number must be above, below, to the left, or to the right of the current number - no diagonal paths are allowed.

53	52			47						211			202	201
54	51						21						203	200
			44								216			
		60			39		23		219			206		
63					40	29	24	15	10					197
			69	70	71				11	8	225			
				73		27		13		7				
	77		83	84			89			92	5		191	
				85		99		95		93				
			119	118	101				165	166	169			
129					102	103	104	105	164					185
		124			115		111		163			172		
			142								160			
136	135						109						179	178
137	138			145						153			176	177

Unsolved Numbrix puzzle (mathinenglish.com)

53	52	49	48	47	34	33	20	19	212	211	210	209	202	201
54	51	50	45	46	35	32	21	18	213	214	215	208	203	200
55	58	59	44	37	36	31	22	17	218	217	216	207	204	199
56	57	60	43	38	39	30	23	16	219	220	221	206	205	198
63	62	61	42	41	40	29	24	15	10	9	222	223	196	197
64	67	68	69	70	71	28	25	14	11	8	225	224	195	194
65	66	75	74	73	72	27	26	13	12	7	6	1	192	193
78	77	76	83	84	87	88	89	90	91	92	5	2	191	190
79	80	81	82	85	86	99	98	95	94	93	4	3	188	189
128	127	120	119	118	101	100	97	96	165	166	169	170	187	186
129	126	121	122	117	102	103	104	105	164	167	168	171	184	185
130	125	124	123	116	115	112	111	106	163	162	161	172	183	182
131	132	133	142	143	114	113	110	107	158	159	160	173	180	181
136	135	134	141	144	147	148	109	108	157	156	155	174	179	178
137	138	139	140	145	146	149	150	151	152	153	154	175	176	177

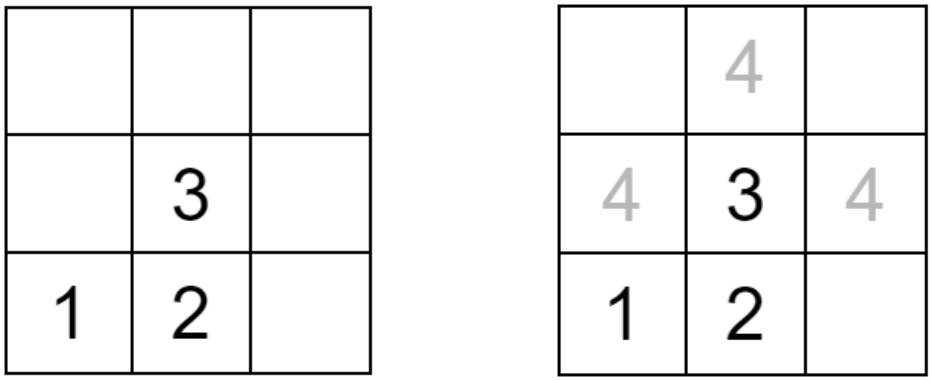
Solved Numbrix puzzle (mathinenglish.com)

The solving algorithm explores potential paths and backtracks upon hitting a dead end via recursion. Its main data structure is an array, `arr`, that holds the contents of the board. As a potential path is explored, the array is filled in, and if a dead end is found, the last number filled in is overwritten with a 0, indicating an open spot. The algorithm indexes into the array when looking for the next number and when checking to see if a spot on the board is already filled.

Inputs to the algorithm include the board to solve and the dimensions of the board. The board is stored as a string, and the dimensions as an int. The algorithm loads the contents of the string into the board array, and outputs the solved board in the array. The main function then prints the unsolved and solved boards, allowing the user to visually inspect the result.

Recursively placing potential paths on the board and backtracking once it's found that the path is invalid is what makes the algorithm computationally expensive. If parallelism could be introduced so that multiple paths can be explored at the same time, this part of the algorithm could benefit greatly. However, the workload makes this tricky; recursive parallelism is not

something we've covered in class, and the large array data structure must be duplicated for each path being explored so that each copy of the board holds a different potential path being explored. One global array will not work, as any board with more than one of a number is invalid, and it would be impossible to determine what numbers were filled in before the path was explored, and what numbers were filled in by processes exploring a different potential path. Thus, the workload is amenable to thread-based parallelization, in which each thread explores a different potential path. This also maximizes locality, as each thread will essentially be assigned a different index of the board array to test, and this index stays constant within the thread. If the index is a valid placement, the next index the thread explores is guaranteed to be above, below, to the left, or to the right of the original index.



Visualization of thread creation to explore all possible positions for next placement. Here, three threads would be created for each of the three possible positions for the number 4.

Approach

As I wanted to use a thread-based approach, I looked into both pthreads and OpenMP. For each language, I sketched out how I would use API calls to parallelize the algorithm. OpenMP's API

had a lot more support for how I wanted to control the launching of threads and how I wanted to handle shared and private data, so that's what I decided to use. I planned on running my code on the GHC machines, as they have GPUs that allow for 8 processes, which works well with what I planned on parallelizing. I later ran my code on PSC as well, which is a GPU with a max of 256 threads.

I mapped the problem to the machine by assigning potential paths to threads (processes). Each time the algorithm finds a new potential path to explore, a new thread is launched via a task (if a thread is not available, the task is assigned to the calling thread or the next thread that completes the task it was assigned). I included a depth parameter that limits how many nested processes can exist - for example, a child thread's child would have a depth of 2. I made my cutoff 8, so that my parallelism is well-suited for the machine.

To implement parallelism, I changed the serial algorithm in a few ways. Instead of returning out of a for loop upon finding a valid path, I had my algorithm set a flag that is checked before the function returns, and created the necessary variables to accomplish this. I also introduced a new variable, `local_arr`. This allows each thread to have a copy of the board array, and work in it without changing the global `arr` or any other thread's board array. If a path is found that completes the board, this copy of `local_arr` is copied into global `arr` and returned.

My project went through many iterations of optimization as I tried to achieve the best speedup. At first, I avoided parallelism in the recursion itself, using only what we had previously used in class: `#pragma omp parallel`, `#pragma omp for`, and `#pragma omp parallel for`. This was in an effort to avoid the sticky situation of learning and implementing parallel recursion, but ultimately, using parallelism solely in data and index calculations did not yield speedup. I began learning about every OpenMP directive, construct, and clause I could, and looking online for information on parallel recursion using OpenMP. I found that `#pragma omp task` and

`#pragma omp single` could be useful in parallelizing my algorithm, and I was able to create a working version of the algorithm that executed recursion in parallel. At first I used an array for any flags set, and was still creating a new copy of `local_arr` every time a task was launched. I found that this copying of memory, and large memory use and access, slowed my algorithm, so I changed the structure of the algorithm slightly to replace these structures with integers or get rid of them altogether. This helped to improve speedup. I also restructured the algorithm so a new thread is launched only after the master checks if the board space it will probe is a valid space and is open, further improving speedup and resource allocation.

I started with prewritten code to solve a numbrix puzzle from rosettacode.org. This code is a sequential solver that I set out to parallelize. I did have to clean up code to dismiss various warnings and errors, as well as fix various style issues and add comments that explain what variables and functions are doing. However, it did work after some minor fixes, so that helped me to focus on approaches to parallelizing the algorithm.

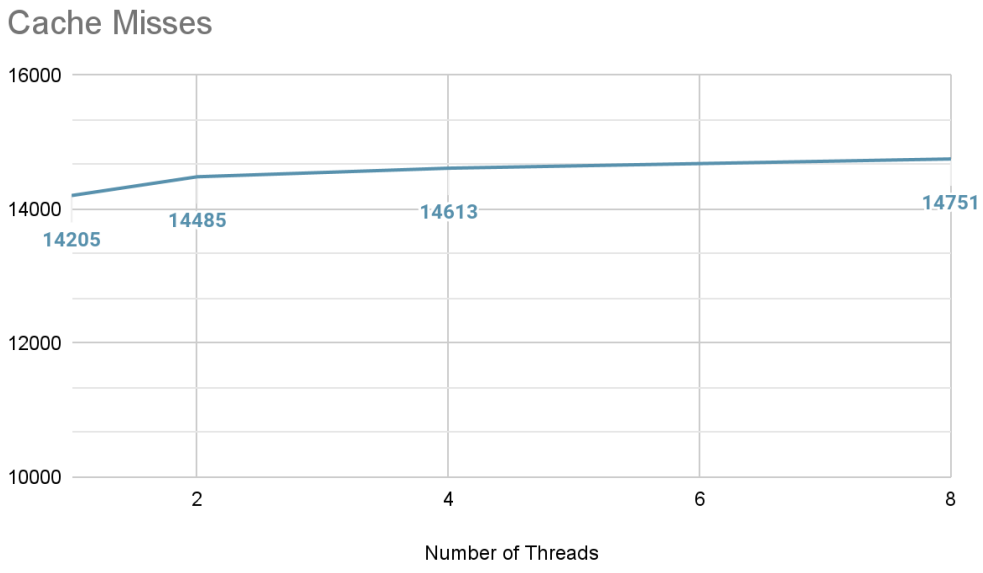
Results

While I was successful in creating a parallel, recursive Numbrix solver, I unfortunately did not meet my goals for speedup. I measured speedup as $\frac{T_1}{T_t}$, where T_1 is the time it takes one thread to solve the board and T_t is the time it takes *t* threads to solve the board (both in milliseconds). I measured speedup for both 9 x 9 and 15 x 15 boards. My speedup was less than one, as the computation time increased as soon as I introduced multiple threads. I analyzed these results alone and with my TA to figure out why this would occur.

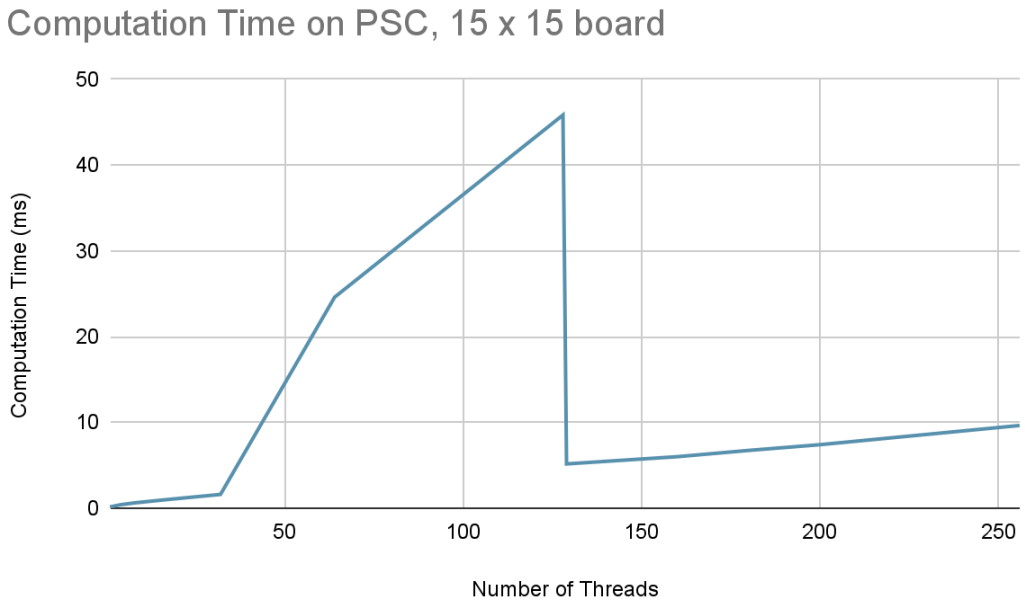
I analyzed the effect of synchronization on my speedup by commenting out the one barrier I had kept in my code, a `#pragma omp taskwait`. Removing it didn't affect the accuracy of my

code, but it also didn't affect speedup. This is likely due to the fact that the OpenMP structured blocks I am using have implicit barriers at the end. Even when adding a `nowait` clause to my `#pragma omp single` call, there is no speedup gained. These implicit barriers cannot be removed, so I will inevitably have some time when threads are idle due to the path they are exploring being shorter than paths other threads are exploring; the threads all have to explore their paths before the algorithm can check flags and return.

I analyzed cache misses as the number of threads increases to see if memory was my bottleneck. Cache misses increasing as thread count increases indicates that memory is a bottleneck, but I found that they stayed pretty much constant. Thus, memory is not my bottleneck. These results make sense: as thread count increases, each thread does the same amount of work per task but has to pick up more tasks. For each task, there shouldn't be many cache misses, as the working set is just the thread's index of the board array, and a small working set reduces capacity and conflict misses. Thus, there should only be a slight increase in cold misses as the new data (for the new task) is loaded in.



After ruling out synchronization and memory bottlenecks, I analyzed computation time in detail on both the GHC and PSC machines to figure out the source of slowdown. Results from the PSC runs mirrored the results from my GHC runs, but showed more detail as I had a maximum of 256 processes, as opposed to 8.



My results show that increasing the number of threads increases computation time, but there was an interesting trend. At $\frac{nproc}{2} + 1$, where *nproc* is the maximum number of processes the machine can support, there is a critical point where the computation time drops off. Before this point, computation time increases rapidly, but after, it begins to increase only slightly as I increase the number of threads. I believe this critical point is where the benefit of more threads begins to mitigate the overhead of thread creation. Spawning a thread and assigning it a task does take some time. Though it is a small amount of time, if the sequential (single-threaded) computation time of the program is small, this overhead could outweigh the benefits of implementing parallelism. This is what I believe is happening in my code; before introducing parallelism, the longest time it took the solver to solve a board was ~0.5 ms for a 15 x 15 board.

After parallelism was introduced, it took the solver only ~2 ms to solve the board using 1 thread. It did seem that speedup improved with a larger board size, but the largest size of Numbrix board is 15 x 15, so I was limited in what I could test my program with. If I were to continue to work on the project, I think it would be interesting to take some time to create a very large Numbrix board, like 100 x 100, and observe speedup trends when solving it.

Overall, the overhead of thread creation and the time it takes to assign tasks outweighs the cost of solving due to the low initial solving time. I have learned from this project that programs best suited for thread-based parallelism take a long time (at least on the order of seconds) to complete, and generally involve large amounts of data. Parallel recursion is possible, and a very interesting topic to explore, but it is best suited for large programs due to the overhead of thread and task creation involved. Further exploration on the topic could include a domain-specific language that easily allows one to parallelize recursion.

References

Original, sequential Numbrix solver

https://rosettacode.org/wiki/Solve_a_Numbrix_puzzle#C.2B.2B

Numbrix Puzzles:

parade.com/numbrix/

https://www.mathinenglish.com/puzzlesnumbrix.php

OpenMP Parallelization:

OpenMP Reference: www.openmp.org/

LLNL HPC OpenMP Tutorial; Author: Blaise Barney, Lawrence Livermore National Laboratory, UCRL-MI-133316. https://hpc-tutorials.llnl.gov/openmp/

OpenMP Parallelization with Recursion

https://stackoverflow.com/questions/40961392/best-way-to-parallelize-this-recursion-using-openmp

https://en.wikibooks.org/wiki/OpenMP/Tasks

Many thanks to the course professors and my advising TA, Ye Lu, for helping me sort out the many bugs and bumps in the road I encountered throughout the course of this project.

Distribution of Credit

I worked alone on this project due to my partner dropping the class. Because of this, I feel that I should get 100% of the credit.