GAME OF LIFE REPORT

MSc Parallel Systems

AUTHORS

- Marios Papamichalopoulos CS310006
- Christos-Charalampos Papadopoulos CS3190009

FILES

- blocks/mpi.c
- blocks/open_mp.c
- blocks/cuda.cu
- series/mpi.c
- funcs.c
- time.h
- mpiP (directory)

In mpiP you can find the mpiP outputs for some of our calculations.

GITHUB

GameOfLife

DESIGN

There is a standard procedure for all the programs. First of all, there is a *TERMINATION_CHECK* macro used in order to perform the *MPI_Allreduce* operation. The *MPI_Allreduce* is called every TERMCHECK_TIMES. Each program was run for *MAX_TIMES* equal to 500.

In addition to the exercise requirements, we have implemented an MPI program with series-block so as to prove practically that block-block scales better. Below we can see the topology used for each type of program. On the left is the block-block variant and on the left the series-block variant.

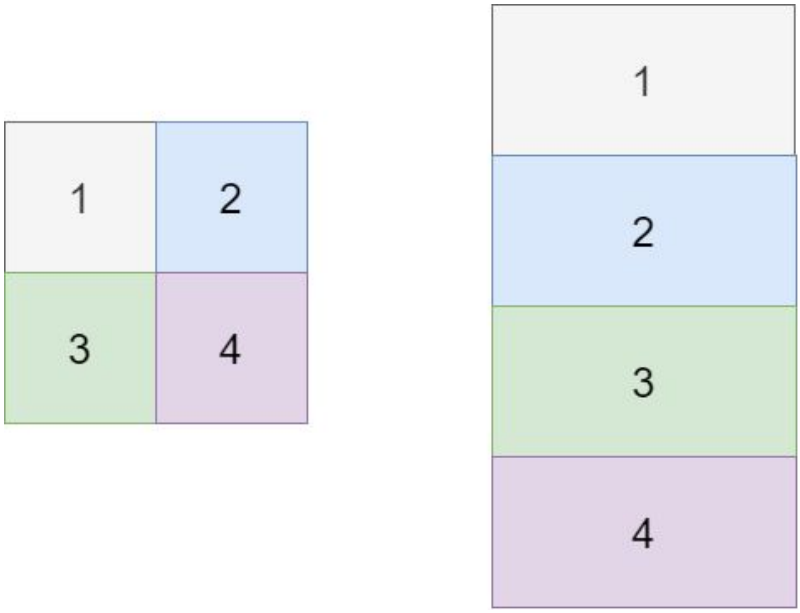

Figure 1: Topology of processes (block-block, series-block)

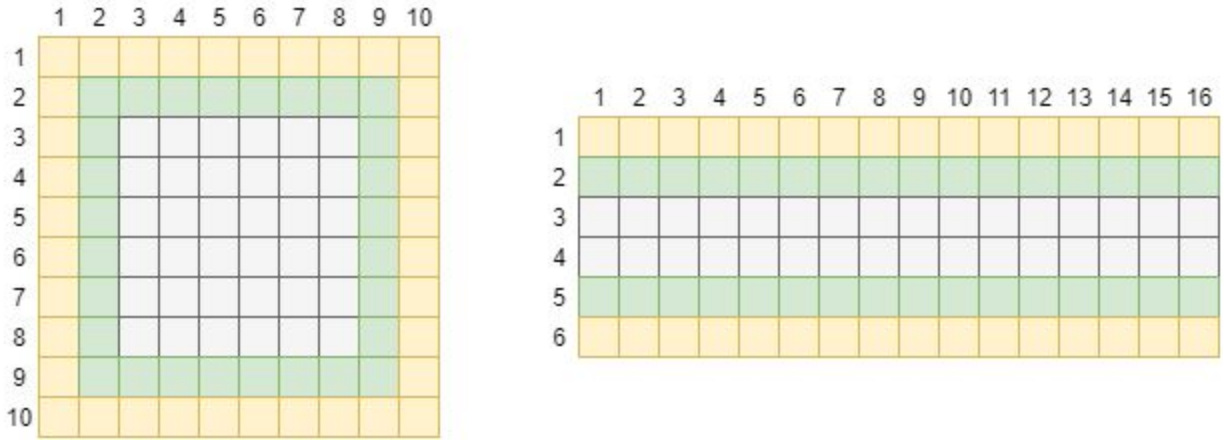

Figure 2: MPI block (block-block, series-block)

The structure of the MPI block depends on the type, as depicted in Figure 2. For block-block as we see from the topology each process communicates with a total of 8 neighbours. (north, east, west, south, north-east, north-west, south-east and south-west). Thus, if we had a program size of **16x16** and **4** processes we would need **4** blocks of **10x10**. The inner part of the block painted green is the actual block. The outer part painted yellow is where the process would keep the blocks received from each neighbour so it could calculate the corresponding game of life criteria.

For the series-block variant, each process needs to communicate with only the south and the north neighbour. Hence, if we again had a program size of **16x16** and **4** processes we would need **4** blocks of **6x16**. The extra **2** rows would be used to store the values from the neighbour.

IMPROVEMENTS

The MPI code was designed in a way to maximize efficiency. We obeyed the instructions given to the letter. Using non-blocking function *ISend* each process sends to the corresponding neighbours the rows and columns they need in order to calculate their inner outer cells (meaning the outer green painted blocks). In addition to this, each process uses non-blocking *Irecv* to receive the blocks missing and assign them to the yellow-painted blocks, in order to calculate the next generation cells. While the non-blocking process takes place each process calculates the cells that do not need any extra information from their neighbours. Looking at Figure 2, these cells are the little square created by (3:4, 3:4) which have a gray color. This reduces idle time greatly. Neighbour ranks are of course calculated outside of the main loop.

There is also a *column datatype* so as to reduce multiple copies as well as a *row datatype* for the MPI series-block program. AllReduce is called every 20 loops. To sum up, our generation cell array and the next generation cell array are 1D and not 2D.

FOSTER METHODOLOGY

1. Theoretical Approach

Foster's methodology describes a way to design and implement parallel solutions of a problem in 4 steps: *Partitioning*, *Communication*, *Agglomeration* and *Mapping*.

Using this method our first step (partitioning) is to find the base (primitive) task of the problem which will then be parallelized. Our base task here is to determine a cell's next state according to its 8 neighbours and our problem consists of a grid containing MxN of these cells.

Next, for the communication step, each cell will send its state to each one of the 8 neighbouring cells and will then receive the states it needs to calculate its own state, 8 in total.

For the agglomeration step, there are different consolidating strategies to merge computations and avoid excess communication which can cause significant overhead. Here we decided to use two design choices to consolidate cells. First we have a block-block strategy which partitions the grid into blocks. Inside a block all the computation is done locally, thus no communication is required. Our other strategy involves creating a Series-Block grid (see Figures 1. And 2.).

Finally, our experiments below belong to the final step of the Foster Methodology (Mapping). Our results for the different parameters, programming platforms and overall choices are described below.

In a few words, what makes the block-block variant better than the series-block is the cells they need to send to their neighbours. In Figure 2, we have a problem of size 16x16. In the block-block version, each process needs to send 6+6+6+6+1+1+1+1 = 28. In the series-block, each process needs to send 16 + 16 = 32. It is common knowledge that the overhead in parallel programs originates from:

- Periods when processors are idle
- Extra computations required by parallel implementation
- Communication time for sending messages

Since, in both programs the idle periods are almost the same (we use *IRecv* & *ISend* and in the meantime calculate the inner cells) and the extra computations required by the parallel implementation are identical, the key difference lies within the communication time that processes spend sending messages. This is not distinctable in low sizes. For example, lookin at the **640x640** program size series-block achieves a time of **294.71 ms** for **64** processes whereas block-block achieves **207.77 ms**.

However, the bigger the program the larger the communication barrier series-block face and this can be deducted from our measurements. For a program size of **5120x5120** and **64** threads, series-block scores **32567.38 ms** and block-block scores **2559.29 ms.** Breaking down the problem, a block-block program would need **64 642x642** blocks. A series-block program would need **64 82x5120** blocks. For each iteration, a block-block program would need to send to its neighbours 638+638+638+638+1+1+1+1 = **2556 cells**. On the contrary, a series-block would need to send to its neighbours 5120+5120 = **10240 cells**, almost 4 times the cells the block-block variant sends and receives. This, also, may have an impact on the process idle time, given that it may wait an arbitrary time period to receive all the cells missing to calculate the generation.

Since *mpptest* was not available in Argo, we took the generic results posted in one thread in e-class. Note that the message transmission time is calculated approximately. Assuming problem with size **5120x5120** and **2** processes the message transmission time is the following:

For **block-block**, given that we have to send 8 messages where neighbours are all in the **same node**:

- $t_{\substack{same\ node \\ message\ transmission\ (1)}} = l + n/b \Rightarrow t_{\substack{same\ node \\ message\ transmission}} \simeq 0.24 + 638/2794.58 = 0.4682\ sec$
- $t_{\substack{same\ node \\ message\ transmission\ (2)}} = l + n/b \Rightarrow t_{\substack{same\ node \\ message\ transmission}} \simeq 0.24 + 1/2794.58 = 0.2403\ sec$

So the **overall time *t*** is:

- $t_{\substack{same\ node \\ message\ transmission}} = 4 * 0.4682 + 4 * 0.2403 = 2.834\ sec$

For **block-block**, given that we have to send 8 messages where neighbours are in **different nodes**:

- $t_{\substack{different\ node \\ message\ transmission\ (1)}} = l + n/b \Rightarrow t_{\substack{different\ node \\ message\ transmission}} \simeq 58.35 + 638/89.34 = 65.49\ sec$
- $t_{\substack{different\ node \\ message\ transmission\ (2)}} = l + n/b \Rightarrow t_{\substack{different\ node \\ message\ transmission}} \simeq 58.35 + 1/89.34 = 58.36\ sec$

So the **overall time *t*** is:

- $t_{\substack{different\ node \\ message\ transmission}} = 4 * 65.49 + 4 * 56.36 = 487.4\ sec$

For **series-block**, given that we have to send 2 messages where neighbours are all in the **same node**:

- $t_{\substack{same\ node \\ message\ transmission}} = l + n/b \Rightarrow t_{\substack{same\ node \\ message\ transmission}} \simeq 0.24 + 5120/2794.58 = 2.07\ sec$

So the **overall time *t*** is:

$t_{\substack{same\ node \\ message\ transmission}} = 2 * 2.07 = 4.14\ sec$

For **series-block**, given that we have to send 2 messages where neighbours are all in **different nodes**:

- $t_{\substack{different\ node \\ message\ transmission}} = l + n/b \Rightarrow t_{\substack{different\ node \\ message\ transmission}} \simeq 58.35 + 5120/89.34 = 115.65\ sec$

So the **overall time *t*** is:

$$t_{\substack{different\ node \\ message\ transmission}} = 2 * 115.65 = 231.3\ sec$$

From the above calculations it is obvious that the time that it takes for the series-block program to send to its neighbours is almost twice the block-block's.

2. Practical Approach

In order to prove the fact that MPI series-block is worse than MPI block-block we actually created a MPI series-block program and ran it for the same amount of processes and size.

Following are the time, speedup and efficiency. In the *MPI & OpenMP* section one can find the respective time, speedup and efficiency for MPI block-block. Compared to block-block the speedup is relatively worse. Also, as the problem size increases the speedup does not increase greatly. For example, in MPI block-block program with size *5120x5120* and *64* processes the speedup **45.8936** whereas in series-block it is **3.6065**. MPI block-block is faster by an order of 15!

MPI series-block - Time (msec)					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	459.58	1806.96	7344.94	29168.22	117455.19
2	382.90	1519.45	6349.08	25699.59	102045.81
4	274.61	1085.26	4382.82	18084.01	73418.83
8	196.35	766.41	3407.38	14565.30	58242.26
16	149.55	561.98	2306.84	10342.00	50895.12

32	115.00	404.96	1559.71	7335.90	46326.54
64	88.42	294.71	1107.47	5212.67	32567.38

MPI series-block - Speedup					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
2	1.2002	1.1892	1.1568	1.1349	1.1510
4	1.6735	1.6650	1.6758	1.6129	1.5997
8	2.3406	2.3576	2.1555	2.0025	2.0166
16	3.0730	3.2153	3.1839	2.8203	2.3077
32	3.9963	4.4620	4.7091	3.9769	2.5353
64	5.1976	6.1313	6.6321	5.5956	3.6065

MPI series-block - Efficiency					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
2	0.6001	0.5946	0.5784	0.5674	0.5755
4	0.4183	0.4162	0.4189	0.4032	0.3999
8	0.2925	0.2947	0.2694	0.2503	0.2520
16	0.1920	0.2009	0.1989	0.1762	0.1442

32	0.1248	0.1394	0.1471	0.1245	0.0792
64	0.0812	0.0958	0.1036	0.0874	0.0563

MPI & OpenMp

We did not measure programs with bigger size than 5120x5120 since it took over 1 minute for the sequential program to finish. In the assignment, it was asked not to measure programs who took longer than 20 seconds to finish as it would be a waste of resources. The number of iterations for all the measurements was set to 500.

The calculations which are painted with a subtle green color are the ones whose mpiPs we have included in our project folder.

1. MPI

MPI block-block - Time (msec)					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	459.58	1806.96	7344.94	29168.22	117455.19
2	236.73	915.89	3796.97	14869.68	59007.41
4	74.71	465.75	1818.82	7517.03	29562.16
8	46.40	240.67	924.25	5123.56	26405.33
16	109.90	126.19	525.79	2455.56	16713.12
32	177.35	198.33	315.22	1013.58	8022.76
64	177.21	207.77	192.69	520.52	2559.29

MPI block-block - Speedup					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
2	1.9413	1.9729	1.9344	1.9615	1.9905
4	6.1515	3.8796	4.0328	3.8802	3.9731
8	9.9047	7.5080	7.9469	5.6929	5.6905
16	4.1818	14.3193	13.9693	11.8784	7.0277
32	2.5913	9.1108	23.3009	28.7774	14.6402
64	2.5934	8.6969	38.1179	56.0366	45.8936

MPI block-block - Efficiency					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
2	0.9706	0.9864	0.9672	0.9807	0.9952
4	1.5378	0.9699	1.0095	0.97	0.9932
8	1.2380	0.9385	0.9933	0.7116	0.7113
16	0.2613	0.8949	0.8730	0.7424	0.4392
32	0.0809	0.2847	0.7281	0.8992	0.4575
64	0.0405	0.1358	0.5955	0.8755	0.7170

2. MPI AllReduce

MPI block-block AllReduce - Time (msec)					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	459.58	1806.96	7344.94	29168.22	117455.19
2	243.17	942.34	4037.40	15860.52	63042.88
4	78.96	479.07	1934.58	8106.65	38165.51
8	47.35	249.38	953.94	5680.22	32659.28
16	126.82	163.44	550.61	2579.63	18691.91
32	207.55	215.02	358.28	1063.95	9144.77
64	215.50	225.86	238.01	606.83	2755.12

MPI block-block AllReduce- Speedup					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
2	1.8899	1.9175	1.8192	1.8390	1.8631
4	5.8204	3.7718	3.7966	3.5980	3.0775
8	9.7060	7.2458	7.6995	5.1350	3.5963
16	3.6238	11.0558	13.3396	11.3071	6.2837
32	2.2140	8.4036	20.5005	27.4150	12.8439
64	2.1326	8.0003	30.8597	48.0665	42.6316

MPI block-block AllReduce - Efficiency					
processes	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
2	0.9449	0.9587	0.9096	0.9195	0.9315
4	1.4551	0.9429	0.9491	0.8995	0.7693
8	1.2132	0.9057	0.9624	0.6418	0.4495
16	0.2264	0.6909	0.8337	0.7066	0.3927
32	0.0691	0.2626	0.6406	0.8567	0.4013
64	0.0333	0.1250	0.4821	0.7510	0.6661

3. MPI & Openmp AllReduce

For the Openmp, we conducted a series of tests on one node using different threads, processes and scheduling choices (static and dynamic with chunks). Our results are presented below. For the runs that gave really bad performances, we omitted the results as we did not test them to avoid wasting resources. “Coll. off” indicates that we don’t use collapse on the double for loop.

MPI & Openmp One Node Runs (msec)					
processes /threads	320x320	640x640	1280x1280	2560x2560	5120x5120
1	459.58	1806.96	7344.94	29168.22	117455.19
1/8	646.49	2578.62	10559.93	x	x
Coll. off	558.32	2379.16	9691.04	x	x
1/16	651.43	2563.23	10537.04	x	x

Coll. off	590.21	2349.56	9251.37	x	x
2/4	331.72	1295.44	5472.57	21542.66	x
Coll. off	283.23	1108.46	5011.37	19758.15	x
2/8	330.48	1299.47	5441.31	21554.43	x
Coll. off	250.55	1188.11	5000.58	19784.12	x
4/2	172.52	657.4	2660.41	10872.12	44724.25
Coll. off	102.74	603.93	2446.37	9986.56	40231.25
4/4	171.9	658.03	2661.85	10871.34	43166.32
Coll. off	121.34	604.92	2440.83	9991.24	39557.1

MPI & Openmp One Node Runs - Speedup					
processes /threads	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
1/8	0.7109	0.7007	0.6955	x	x
Coll. off	0.8231	0.7595	0.7579	x	x
1/16	0.7055	0.7050	0.6971	x	x
Coll. off	0.7787	0.7691	0.7939	x	x
2/4	1.3854	1.3949	1.3421	1.3540	x
Coll. off	1.6226	1.6302	1.4657	1.4763	x
2/8	1.3906	1.3905	1.3498	1.3532	x
Coll. off	1.8343	1.5209	1.4688	1.4743	x
4/2	2.6639	2.7486	2.7608	2.6828	2.6262
Coll. off	4.4732	2.9920	3.0024	2.9207	2.9195
4/4	2.6735	2.7460	2.7593	2.6830	2.7210
Coll. off	3.7875	2.9871	3.0092	2.9194	2.9693

MPI & Openmp One Node Runs - Efficiency					
processes /threads	320x320	640x640	1280x1280	2560x2560	5120x5120
1	1	1	1	1	1
1/8	0.0889	0.0876	0.0869	x	x
Coll. off	0.1029	0.0949	0.0947	x	x
1/16	0.0441	0.0441	0.0436	x	x
Coll. off	0.0487	0.0481	0.0496	x	x
2/4	0.1732	0.1744	0.1678	0.1692	x
Coll. off	0.2028	0.2038	0.1832	0.1845	x
2/8	0.0869	0.0869	0.0844	0.0846	x
Coll. off	0.1146	0.0951	0.0918	0.0921	x
4/2	0.3330	0.3436	0.3451	0.3354	0.3283
Coll. off	0.5592	0.3740	0.3753	0.3651	0.3649
4/4	0.1671	0.1716	0.1725	0.1677	0.1701
Coll. off	0.2367	0.1867	0.1881	0.1825	0.1856

We selected the node setup with 4 processes and 2 threads per process without collapse on the double for loop as it is the one with the best efficiency (but still pretty low).

Next, we conduct experiments with a dynamic schedule and different chunk sizes.

MPI & Openmp - Dynamic Schedule for Best Node Setup(msec)					
Chunk size	320x320	640x640	1280x1280	2560x2560	5120x5120
Static	102.74	603.93	2446.37	9986.56	40231.25
50	154.87	592.85	2325.37	10345.77	39209.04
100	152.54	591.69	2414.2	9891.87	41355.07
200	154.33	592.64	2404.08	9891.96	39227.5
400	152.3	592.14	2405.04	9895.51	41608.33

MPI & Openmp - Dynamic Schedule for Best Node Setup - Speedup					
Chunk size	320x320	640x640	1280x1280	2560x2560	5120x5120
Static	4.4732	2.9920	3.0024	2.9207	2.9195
50	2.9675	3.0479	3.1586	2.8193	2.9956
100	3.0128	3.0539	3.0424	2.9487	2.8402
200	2.9779	3.0490	3.0552	2.9487	2.9942
400	3.0176	3.0516	3.0540	2.9476	2.8229

MPI & Openmp - Dynamic Schedule for Best Node Setup - Efficiency					
Chunk size	320x320	640x640	1280x1280	2560x2560	5120x5120
Static	0.5592	0.3740	0.3753	0.3651	0.3649
50	0.3709	0.3810	0.3948	0.3524	0.3745
100	0.3766	0.3817	0.3803	0.3686	0.3550
200	0.3722	0.3811	0.3819	0.3686	0.3743

400	0.3772	0.3814	0.3817	0.3685	0.3529

We didn't notice any significant changes between the two schedules and different chunk sizes so we decided to use a dynamic schedule with 50 chunk size due to the fact that it has the best speedup and efficiency at the 5120x5120 grid.

Finally we present our experiments for testing on 2, 4, 8, and 10 nodes on the "Argo" cluster.

MPI & Openmp - Scaling of best node Setup (msec)					
Nodes/Total Threads	320x320	640x640	1280x1280	2560x2560	5120x5120
Static	102.74	603.93	2446.37	9986.56	40231.25
2/16	165.84	371.86	1241.45	4918.58	20852.62
4/32	235.44	236.08	2318.97	2529.62	10141.49
8/64	185.77	185.25	401.86	1277	5329.84
10/80	4206.35	4160.4	4237.72	4106.5	4127.48

MPI & Openmp - Scaling of best node Setup - Speedup					
Nodes/Total Threads	320x320	640x640	1280x1280	2560x2560	5120x5120
Static	1	1	1	1	1
2/16	2.7712	4.8592	5.9164	5.9302	5.6326
4/32	1.9520	7.6540	3.1673	11.5307	11.5817
8/64	2.4739	9.7542	18.2774	22.8412	22.0373
10/80	0.1093	0.4343	1.7332	7.1029	28.4569

MPI & Openmp - Scaling of best node Setup - Efficiency					
Nodes/Total Threads	320x320	640x640	1280x1280	2560x2560	5120x5120
Static	1	1	1	1	1
2/16	0.1732	0.3037	0.3698	0.3706	0.3520
4/32	0.0610	0.2392	0.0990	0.3603	0.3619
8/64	0.0387	0.1524	0.2856	0.3569	0.3443
10/80	0.0014	0.0054	0.0217	0.0888	0.3557

As we can see, the efficiency is pretty low as we use a large number of threads and don't get very good results. Pure MPI implementation is way better with efficiency ranging from 59%-87% as opposed to 28%-35% for the Openmp hybrid.

CUDA

In our final experiment, we created a “pure” cuda program and tested its performance.

CUDA Runtime and Speedup					
	320x320	640x640	1280x1280	2560x2560	5120x5120
Static	102.74	603.93	2446.37	9986.56	40231.25
CUDA	228.38	582.95	1861.84	5002.74	27062.09
Speedup	2.0123	3.0997	3.9450	5.8304	4.3402

We observe that the CUDA performance gets bottlenecked by the time it takes to actually copy from CPU to GPU. In our 5120x5120 grid, the program report showed that 99.20% of the total time (27.0539s) was used for copying to device and in the GPU, 52.34% (14.0071s) and 43.37% (11.6055s) were used to copy from Host to Device and from Device to Host respectively.

CONCLUSION

We have answered all the questions required by the exercise. We provided both a theoretical and a practical approach on why block-block organization is better than series-block.

At last, we presented meticulous calculations performed on Argo for MPI, MPI + OpenMP and CUDA along with some remarks.