Gabriel Mancino-Ball
mancig@rpi.edu

Parallel Computing Assignment 3

CSCI 6360
March 10, 2021

1 Executing the Code

Please note the following file name changes from what was listed in the homework outline:

1. `highlifeMpi.c` is the `C` file containing all of the MPI code
2. `highlifeCuda.cu` is the `CUDA` file containing all of the CUDA code.

To properly run `highlifeMpi.c`, the following arguments are required:

1. the world configuration (0, 1, 2, 3, 4, 5),
2. the world size (a power of 2),
3. the number of world updates/iterations,
4. **and** the thread count.

Additionally, with the addition of MPI, `mpirun -np <num_gpus>` must be proceed the execution of the program, thus an example of proper execution of the code is

```
mpirun -np 4 ./highlife-exe 5 16384 128 256
```

1.1 Printing the Worlds

For all world sizes smaller than 64×64 (e.g. 32×32), the initial state of the world for MPI rank n will be printed to the file `initial_n.txt` and the final state (i.e. after all iterations have been performed) will be printed to the file `output_n.txt`. This allows for error checking and making sure the code works on smaller worlds. **If you plan to check the output of the code for larger worlds, change lines 137 and lines 193 to be the size of the world you desire**.

2 Numerical Results

We test the MPI/CUDA linked program on a world size of 16382×16384 for 128 iterations with a thread size of 256 and MPI ranks from the set $\{1, 2, 3, 4, 5, 6, 12\}$. We report the *relative speedup factor* which is computed as

$$\text{speedup factor}(t_g, t_1, n) = \frac{|t_g - nt_1|}{nt_1} \cdot 100 \tag{1}$$

where n is the number of GPUs used, t_g is the execution time for MPI code, and t_1 is the execution time for a single GPU. **Notice**, we multiply be n here since a single GPU execution contains $\frac{1}{n}$ proportion of the world size in the MPI version. Figure 1 shows our results. For each GPU size, we perform 3 runs and take the average execution time.

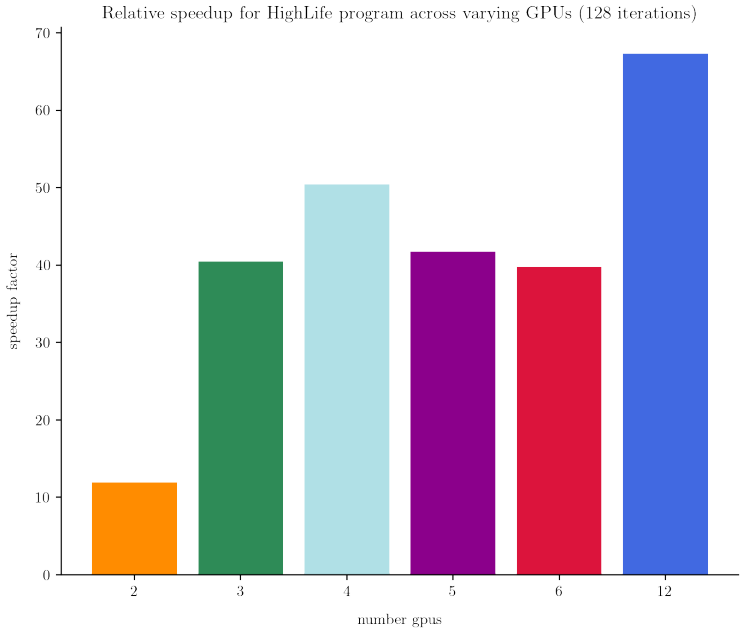

Figure 1: Relative speedup performance computed by (1) on world size 16384×16384 for 128 iterations.

Another important metric to consider is the number of cells updated per second. Table 1 demonstrates these results, indicating the best performance in bold face.

Number of GPUs	1	2	3	4	5	6	12
Cells Updated/Second	6.31E+11	7.16E+11	1.06E+12	1.27E+12	1.08E+12	1.05E+12	**1.93E+12**

Table 1: Cells updated per second. Bold indicates the best performance.

2.1 Conclusions

Figure 1 demonstrates clearly a superior MPI configuration; 12 GPUs outperforms all of the other possible MPI ranks significantly. For the other MPI configurations, it seems that 4 GPUs has superior performance; I think this occurs because the `MPI_Isend` and `MPI_Irecv` functions require an `MPI_Wait` function to ensure that each GPU participating in the communication has sent and received its data properly. As we have more and more GPUs, this waiting process could take a longer time to complete because global synchronization needs to occur after every send and receive. However, at some point, when the world becomes very large (i.e. as the number of GPUs increases) we can see that MPI lends a significant speed up over just a single GPU case.

2.1.1 Additional Experiments

We perform two additional experiments: 1) 1024 iterations on the 16384×16384 world size and 2) 128 iterations on a 65536×65536 world size. For both of these experiments, we do not include the 12 GPU case (as AiMOS resources were not required for this part of the assignment). Figures 2 and 3 show the results, respectively. We note that as the world size gets larger, MPI performs poorly (since it has to send more data) as evident in Figure 3. For all of these experiments, we only ran the simulation 1 time for each configuration; employing an averaging strategy may yield different results.

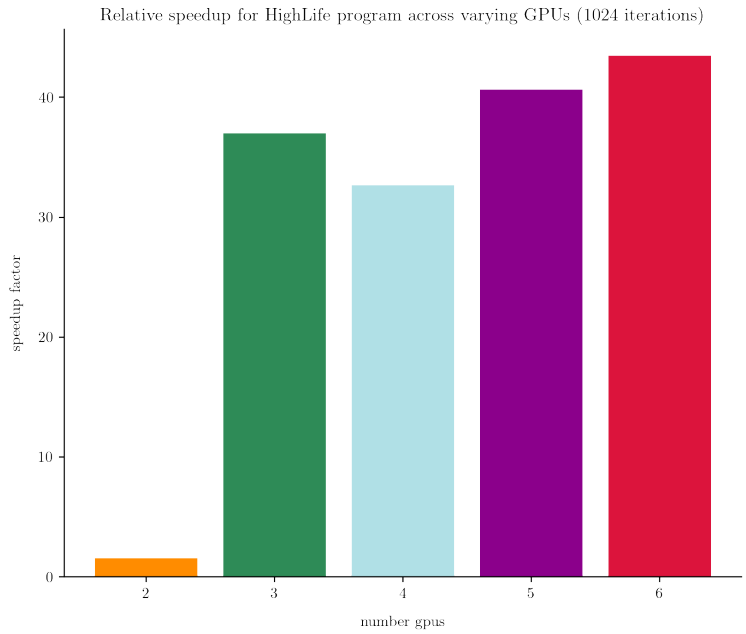

Figure 2: Relative speedup performance computed by (1) on world size 16384×16384 for 1024 iterations.

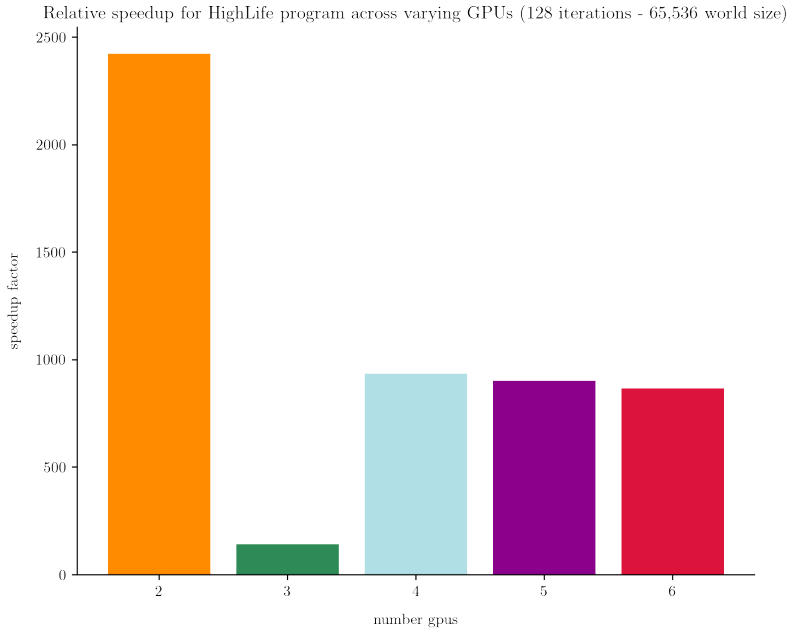

Figure 3: Relative speedup performance computed by (1) on world size 65536×65536 for 128 iterations.

Multiple Nodes and Multiple GPUs

For sake of completeness, we include the modified `slurmSpectrum.sh` file and the command to run a script across multiple compute nodes (this is just for future reference). The command to run a script across multiple nodes is

```
sbatch -N <num\_nodes> --ntasks-per-node=<num\_gpus> --gres=gpu:<num\_gpus>  -t
 <hours:minutes:seconds> ./slurmSpectrum.sh
```

which executes the following script (`slurmSpectrum.sh`):

```
#!/bin/bash -x

if [ "x$SLURM_NPROCS" = "x" ]
then
    if [ "x$SLURM_NTASKS_PER_NODE" = "x" ]
    then
        SLURM_NTASKS_PER_NODE=1
    fi
    SLURM_NPROCS=‘expr $SLURM_JOB_NUM_NODES \* $SLURM_NTASKS_PER_NODE‘
    else
        if [ "x$SLURM_NTASKS_PER_NODE" = "x" ]
        then
            SLURM_NTASKS_PER_NODE=‘expr $SLURM_NPROCS / $SLURM_JOB_NUM_NODES‘
        fi
fi


srun hostname -s | sort -u > /tmp/hosts.$SLURM_JOB_ID
awk "{ print \$0 \"-ib slots=$SLURM_NTASKS_PER_NODE\"; }" /tmp/hosts.$SLURM_JOB_ID > /tmp/tmp.$SLURM_JOB_ID
mv /tmp/tmp.$SLURM_JOB_ID /tmp/hosts.$SLURM_JOB_ID

# LOAD MODULES/FILES
conda activate <env>
module load xl_r spectrum-mpi cuda/10.2

mpirun -hostfile /tmp/hosts.$SLURM_JOB_ID -np $SLURM_NPROCS /gpfs/u/home/<Project>/<Project_user>/<barn or scratch>/<path to file> params
rm /tmp/hosts.$SLURM_JOB_ID
```