# Homework 4

Abhijit Chowdhary

April 15, 2019

## Matrix Vector Operations on a GPU

To complete these, notice that if we have a function $f : \mathbb{R}^n \to \mathbb{R}$ that computes the summation reduction of a vector, then if we want to dot product between two vectors we can simply call $f(\mathbf{a} \odot \mathbf{b})$, where $\odot$ is the elementwise product of the matrices. Similarily, if we want to compute the matrix vector product, we can view that as repeated dot products of the elements of the matrix. I imagine this last approach is very naive, which results in the very poor bandwidth of the computation shown below. The dot product is done on vectors of size $2^{25}$ and the matrix vector product on a matrix of size $2^{12} \times 2^{12}$.

| NVIDIA Hardware | Dot Product Bandwidth | MVec Bandwidth |
|---|---|---|
| Tesla V100-SXM2-16GB | 822.563337 GB/s | 22.577228 GB/s |
| Tesla PC100-PCIE-16GB | 452.578481 GB/s | 12.478081 GB/s |
| GeFore RTX 2080 Ti | 286.316796 GB/s | 08.704399 GB/s |
| GeFore GTX Titan Z | 101.127573 GB/s | 00.167960 GB/s |

This last GPU was from a system that was also running other jobs, and thus is likely why it is so much slower. I agree that slurm would be nice on the cuda servers.

## 2D Jacobi method on a GPU

Here I implemented Jacobi iteration naively on a GPU, just running each Jacobi step on the GPU. The kernel written looks like:

```
__global__ void jacobi_step_gpu(
                double *u, const double *u0, const double *f,
                const long N)
{
  int idx = (blockIdx.x) * blockDim.x + threadIdx.x + 1;
  int jdx = (blockIdx.y) * blockDim.y + threadIdx.y + 1;
  double h = 1.0 / (double)N;
  u[idx*N + jdx] = 0.25 * ( h*h*f[idx*N + jdx] + u0[(idx-1)*N + jdx] +
                                                 u0[idx*N + (jdx-1)] +
                                                 u0[(idx+1)*N + jdx] +
```

```
                                                    u0[idx*N + (jdx+1)] );
}
```

This results in a pretty good speedup for the code, relative to the version written in OpenMP. For the problem where we have a system of size $2^{12} \times 2^{12}$ with 10000 Jacobi iterations we find that it takes:

| NVIDIA Hardware | Runtime |
|---|---|
| Tesla V100-SXM2-16GB | 12.6551s |
| Tesla PC100-PCIE-16GB | 16.3621s |
| GeFore RTX 2080 Ti | 14.7191s |

# Pitch your final project

For my final project, I plan on working solo to try and implement Parareal, a numerical ODE system solver which is parallel in time. The basic idea behind the solver is to solve the system with a course and inexpensive solver, and use it's answers to correct in parallel with a finer solver. Repeat this process until desired stopping point. With this final project, I intend to:

- Implement as efficiently as possible with OpenMP (and CUDA/MPI time and/or method permitting) and verify the correctness of the method.
- Test it on a few different model ODEs, and solve a parabolic or elliptic PDE (discretized with the method of lines approach), noting efficiency.
- Experiment with hardware, verifying theoretical estimates for speedup relative to resources given.
- Experiment with choices of fine and course solvers, with the intent of getting the fastest solution for a given accuracy.

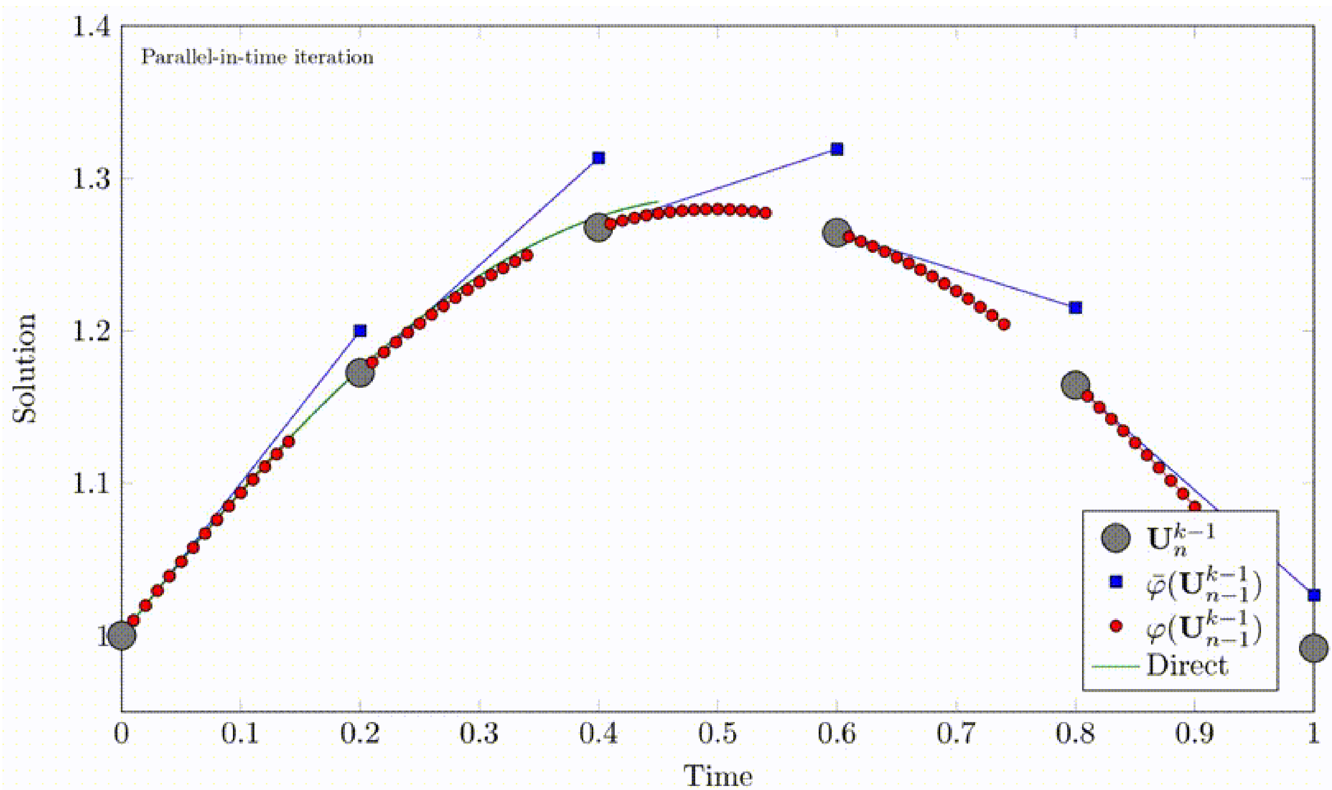


Figure 1: A sample Parareal solve mid iteration. Check out parareal.gif for gif version for the above (very slow). Credits wikipedia.