Project 3: Data Parallelism on Distributed Memory Platforms using MPI

Dustin (Ting-Hsuan) Ma

I. Introduction

Message Passing Interface (MPI) is a message passing standard created for parallel computing architectures. Parallel computing uses MPI to allow the messages between graphics processing unit (GPU) to communicate with one another. Using the parallel computing techniques, total wall time comparison is necessary to demonstrate the benefits of parallel computing. Strong and weak scaling is essential to understand the advantage of MPI parallel programming with a large problem size.

Using finite difference (FD) method, the steady state temperature distribution was implemented into CPU and MPI. The problem consist of a four sided square with temperatures of 100°C, 0°C, 0°C, 0°C, assigned to North (upper edge), South (lower edge), East (right edge), West (left edge), respectively.

The forward marching method is used to iterate to a time step of the user's desired end time. The Courant number (CFL) is set to 0.25 to avoid simulation blow up. In this code, North is given as the bottom most edge due to the origin of the i and j index being the upper right corner in a given mesh. Therefore maximum i, and j index is at the bottom edge. This does not effect the visualization of the output.

II. Method

Heat conduction is governed by,

$$\frac{\partial \theta}{\partial t} = \frac{\partial^2 \theta}{\partial x^2} + \frac{\partial^2 \theta}{\partial y^2} \tag{1}$$

The allowable CFL number from Eq.1 is set to $CFL = 0.25$.

A finite-difference of heat conduction approximation could be applied to iterate over time with,

$$\begin{aligned}\frac{{\theta_{i,j}}^{t+1} - {\theta_{i,j}}^{t}}{\partial t} &= (\frac{{\theta_{i+1,j}}^{t} - 2{\theta_{i,j}}^{t} + {\theta_{i-1,j}}^{t}}{\partial x^2} \\ &+ \frac{{\theta_{i,j+1}}^{t} - 2{\theta_{i,j}}^{t} + {\theta_{i,j-1}}^{t}}{\partial y^2})\end{aligned} \tag{2}$$

The steady-state analytical solution can be calculated with the following series,

$$\theta(x,y) = \sum_{n=1,3,5,\ldots}^{\infty} \frac{4\theta_N}{n\pi} \frac{sin(\frac{n\pi x}{L}) sinh(\frac{n\pi y}{L})}{sinh(n\pi)} \tag{3}$$

The analytical solution does not include an iteration value $n > 101$.

A. CPU

In the serial code, programming techniques such as pointer swapping, and function utilization were used.

B. MPI

Functions utilized in the program are reference from [1],

- MPI_Init
- MPI_Finalize
- MPI_Scatter
- MPI_Gather
- MPI_Cart_create
- MPI_Cart_shift

C. Scalability

Amdahls Law states that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited regardless of the number of cores available. Scalibility Definition:

- In general, a problem is **scalable** if it can handle ever increasing problem sizes.
- Increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is **strongly scalable**.
- Keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is **weakly scalable**.

1) Strong Scaling: (ex. Figure 1) is defined as how the solution time varies with the number of processors for a fixed total problem size. Strong scaling came from the question, "Will it take less computation time if processes are increased?". Strong scaling was done using a larger problem of 100x100 matrix, computation wall times are collected for 2, 4, 8,and 16 processes.

2) Weak Scaling: (ex. Figure 2) is define as how the solution time varies with the number of processors for a fixed problem size per processor. Weak scaling came from asking the question, "What is the best performance I can get out of one process?". Weak scaling test was done by running 1000 points on 1 process, 2000 points on 2 processes. Theoretically these two runs should output the same time.

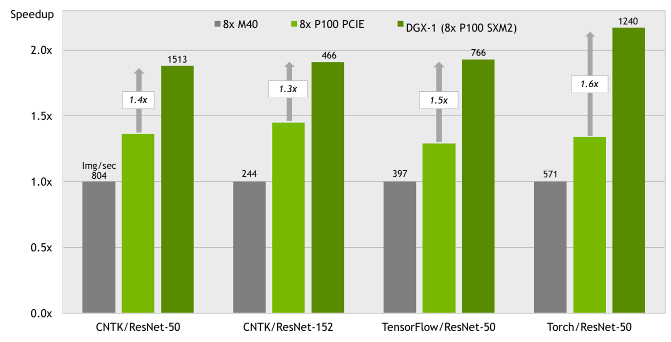

Fig. 1. Speedup using all 8 Tesla P100s of DGX-1 vs. 8-GPU Tesla M40 and Tesla P100 systems using PCI-e interconnect for the ResNet-50 and Resnet-152 deep neural network architecture on the popular CNTK (2.0 Beta5), TensorFlow (0.12-dev), and Torch (11-08-16) deep learning frameworks. [2]

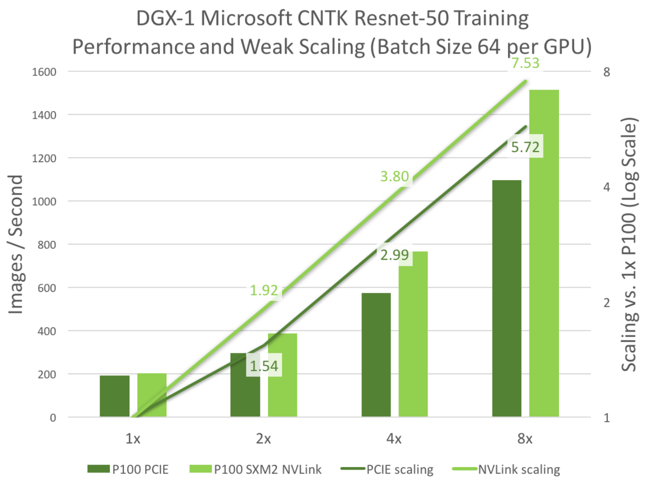

Fig. 2. The bars present performance on one, two, four, and eight Tesla P100 GPUs in DGX-1 using NVLink for inter-GPU communication (light green) compared to an off-the shelf system with eight Tesla P100 GPUs using PCIe for communication (dark green) [2]

III. Result

A. Analytical

In a 20x20 cm square, the edges are subdivided into 10 points. Iteration of the analytical uses Eq.3 to compute the summation of each iteration point with a given x-y coordinate. The code is implemented to solve the 2D heat conduction over any size square with a user desired mesh resolution. Figure 3 shows the steady-state solution to a heat conduction with three of the squares edges set to 0°C, and the North edge set to 100°C.

B. CPU

The results for heat conduction in a 2D square showed that minor differences exists between the analytical and numerical solution. The analytical is only accurate to a summation of $n < 101$, therefore values after does not

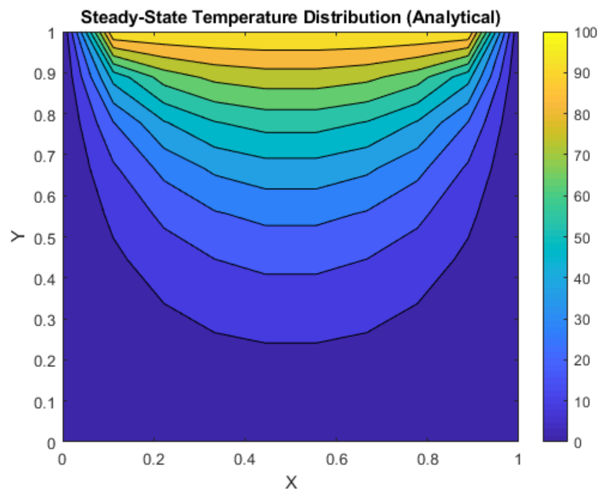

Fig. 3. Analytical solution to the steady-state temperature distribution

apply to the analytical solution. Figure 3, and Figure 4 shows the results for the analytical and numerical solution.

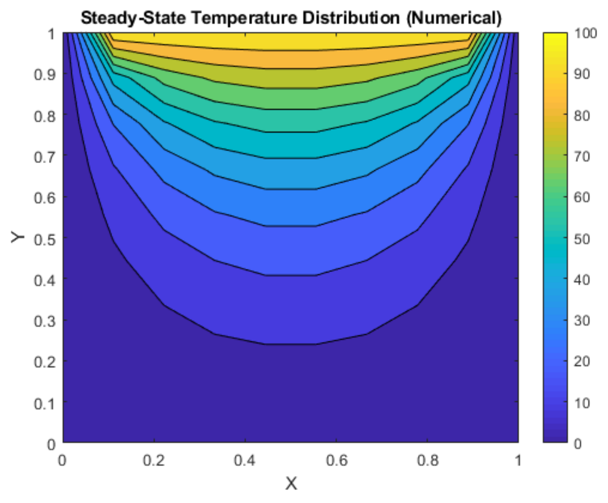

Fig. 4. Numerical solution to the steady-state temperature distribution

C. MPI

While running the MPI portion of the code, the solver does not propagate the solution to the rest of the processes. Results only existed in process 0 while using a $process > 1$. The debugging process was to fill the matrix with incremental values from zero to size of array before scattering. After Scattering, print out data of the local array. when local array has the correct amount of elements, begin sending and receiving data from source and destination process. To make sure this step was done correctly, each process printed out its individual local array for visual inspection. After certain amount of iteration, the data is gather back to another matrix with the same size as the initial scattered matrix while excluding the ghost cells. Section **??** shows a small case of a 10x10 2D mesh running with 2 processes.

D. Scalibility

Scalibility was not performed because the MPI code is not working properly.

IV. Conclusion

The finite difference method have been implemented to run on both CPU and MPI. The MPI portion was a failure, thus running a scaling test was not possible. The serial code runs with accurate approximation for a small problem size. Functions from the MPI library were used to call the MPI functions.
All code is attached, and color highlighted with $a2ps$ and $ps2pdf$ function in the Linux library .

References

[1] "Deinompi."
[2] "Nvidia dgx-1: The fastest deep learning system," Jun 2018.

```c
/*
 * ME 2054 Parallel Scientific Computing
 * Project 1 - Finite Difference Solution of a Vibrating 2D Membrane on a GPU
 * Due: November 6,2018
 *
 * Author: Dustin (Ting-Hsuan) Ma
 *
 * Compile: gcc cpuHeat.c -O3 -lm -o CPU.exe
 * Clang: clang-format -i cpuHeat.c
 */

#include "timer.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

// Spacial
#define LX (REAL) 20.0f
#define LY (REAL) LX
#define NX (INT) 10
#define NY (INT) NX
#define DX LX / ((REAL) NX - 1.0f)
#define DY (REAL) DX

// Temperature
#define TMAX (REAL) 100.0f
#define TMIN (REAL) 0.0f

// Time
#define DT (REAL) 0.25f * DX *DX

// Calculation index
#define IC i + j *NX
#define IP1 (i + 1) + j *NX
#define IM1 (i - 1) + j *NX
#define JP1 i + (j + 1) * NX
#define JM1 i + (j - 1) * NX

typedef double       REAL;
typedef const double C_REAL;
typedef int          INT;

void SolveHeatEQ(C_REAL *now, REAL *out)
{
        for (INT j = 1; j < NY - 1; j++) {
                for (INT i = 1; i < NX - 1; i++) {
                        out[IC] = (((now[IP1] - 2.0f * now[IC] + now[IM1]) / (DX * DX))
                                   + ((now[JP1] - 2.0f * now[IC] + now[JM1])) / (DY * DY))
                                  * DT
                                  + now[IC];
                }
        }
}

void initializeM(REAL *in)
{
        for (INT j = 0; j < NY; j++) {
                for (INT i = 0; i < NX; i++) {
                        if (j == 0) {
                                in[IC] = TMIN;
                        }
                        if (j == NY - 1) {
                                in[IC] = TMAX;
                        }
                        if (i == 0) {
                                in[IC] = TMIN;
                        }
                        if (i == NX - 1) {
                                in[IC] = TMIN;
                        }
                }
        }
}

void analyticalSolution(REAL *out, C_REAL *x, C_REAL *y)
{
        REAL part = 0.0f;
        for (INT i = 1; i < NX - 1; i++) {
                for (INT j = 1; j < NY - 1; j++) {
                        for (REAL n = 1.0f; n <= 101.0f; n += 2.0f) {
                                REAL a = 4 * TMAX / (n * M_PI); // shows good values
                                REAL b = sin(n * M_PI * x[IC] / LX);
                                REAL c = sinh(n * M_PI * y[IC] / LY);
                                REAL d = sinh(n * M_PI);
                                part += (a * b * c / d);
```

```c
                        }
                        out[IC] = part;
                        part    = 0.0f;
                }
        }
}

void meshGrid(REAL *xGrid, REAL *yGrid)
{
        for (INT j = 0; j < NY; j++) {
                for (INT i = 0; i < NX; i++) {
                        xGrid[IC] = i * DX;
                        yGrid[IC] = j * DY;
                }
        }
}

void outputMatrix(C_REAL *in)
{
        for (INT j = 0; j < NY; j++) {
                for (INT i = 0; i < NX; i++) {
                        printf("%8.4f ", in[IC]);
                }
                printf("\n");
        }
        printf("\n");
}

INT main(int argc, char *argv[])
{
        if (argc < 2) {
                perror("Command-line usage: executableName <Number of Iteration>");
                exit(1);
        }
        int ENDTIME = atoi(argv[1]);

        // Allocating memory
        REAL *xGrid      = (REAL *) calloc(NX * NY, sizeof(*xGrid));      // X Grid
        REAL *yGrid      = (REAL *) calloc(NX * NY, sizeof(*yGrid));      // Y Grid
        REAL *theta      = (REAL *) calloc(NX * NY, sizeof(*theta));      // Theta now
        REAL *theta_new  = (REAL *) calloc(NX * NY, sizeof(*theta_new));  // Next Theta
        REAL *analytical = (REAL *) calloc(NX * NY, sizeof(*analytical)); // Analytical

        // Analytical Solution
        meshGrid(xGrid, yGrid);
        initializeM(analytical);
        analyticalSolution(analytical, xGrid, yGrid);
        printf("===== Analytical ===== \n");
        outputMatrix(analytical);

        // Initializing matrix boundaries
        initializeM(theta);
        initializeM(theta_new);

        // Timing
        double start, finish;
        REAL * tmp, time = 0.0f;

        GET_TIME(start);
        while (time < ENDTIME) {
                SolveHeatEQ(theta, theta_new);
                tmp       = theta;
                theta     = theta_new;
                theta_new = tmp;

                time += DT;
        }
        GET_TIME(finish);

        printf("===== Numerical ===== \n");
        outputMatrix(theta_new);

        // Outputing CPU solution
        printf("elapsed wall time = %3.5f (ms)\n", (finish - start) * 1e3);
        printf("\n");

        // Deallocating Memory
        free(theta);
        free(theta_new);
        free(xGrid);
        free(yGrid);
        free(analytical);
        theta      = NULL;
        theta_new  = NULL;
        xGrid      = NULL;
        yGrid      = NULL;
        analytical = NULL;

        return EXIT_SUCCESS;
}
```

```c
/*
 * In this project, I tried to implement parallel programing with the heat equation
 * of a 2D square. I first wrote a serial version of what the Heat equation would be,
 * then I start implementing the MPI parallell programming. Throughout the duration
 * of working with MPI, I have not found the reason to why my solver isn't propegating
 * to the rest of the proccesses. I tested the Send and Receive code block, but when
 * solving the problem, I couldn't get it.
 *
 * Author: Dustin (Ting−Hsuan) Ma
 *
 * To Compile: mpicc −o MPI.exe −lm mpiHeat.c
 * To Run: mpirun −np 4 ./MPI.exe
 *
 */

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

// Spacial
#define LX (REAL) 20.0f
#define LY (REAL) LX
#define NX (INT) 20
#define NY (INT) NX
#define DX LX / ((REAL) NX − 1.0f)
#define DY (REAL) DX

// Temperature
#define TMAX (REAL) 100.0f
#define TMIN (REAL) 0.0f

// Time
#define DT (REAL) 0.25f * DX *DX
#define MAXITER 100

// Calculation index
#define IC i + j *NX
#define IP1 (i + 1) + j *NX
#define IM1 (i − 1) + j *NX
#define JP1 i + (j + 1) * NX
#define JM1 i + (j − 1) * NX

#define MASTER 0

typedef double       REAL;
typedef const double C_REAL;
typedef int          INT;
/*
void initializeM(REAL *a)
{
    for (INT j = 0; j < NY; j++) {
        for (INT i = 0; i < NX; i++) {
            INT idx = i + j * NX;
            a[idx]  = (REAL) idx;
        }
    }
}
*/
void initializeM(REAL *in)
{
        for (INT j = 0; j < NY; j++) {
                for (INT i = 0; i < NX; i++) {
                        if (j == 0) {
                                in[IC] = TMIN;
                        }
                        if (j == NY − 1) {
                                in[IC] = TMAX;
                        }
                        if (i == 0) {
                                in[IC] = TMIN;
                        }
                        if (i == NX − 1) {
                                in[IC] = TMIN;
                        }
                }
        }
}

void decomposeMesh_1D(const int N, const int nProcs, const int myRank, int *start, int *end)
{
        *start = myRank * N / nProcs;
        *end   = *start + N / nProcs;
}

void SolveHeatEQ(C_REAL *now, REAL *out, const int nrow)
{
```

```c
        for (INT j = 1; j < nrow; j++) {
                for (INT i = 1; i < NX − 1; i++) {
                        out[IC] = (((now[IP1] − 2.0f * now[IC] + now[IM1]) / (DX * DX))
                                   + ((now[JP1] − 2.0f * now[IC] + now[JM1])) / (DY * DY))
                                  * DT
                                  + now[IC];
                }
        }
}

void exchange_SendRecv(REAL *in, const int src, const int dest, const int nrow, const int myRank)
{
        int tag0 = 0;                                                           // up send tag
        int tag1 = 1;                                                           // down send tag
        MPI_Send(in + (nrow * NX), NX, MPI_DOUBLE, dest, tag1, MPI_COMM_WORLD); // send to down
        MPI_Send(in + NX, NX, MPI_DOUBLE, src, tag0, MPI_COMM_WORLD);           // send to up

        MPI_Recv(in, NX, MPI_DOUBLE, src, tag1, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // receive from
up
        MPI_Recv(in + ((nrow + 1) * NX), NX, MPI_DOUBLE, dest, tag0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE); // receive from down
}

void outputMatrix(C_REAL *in)
{
        for (INT j = 0; j < NY; j++) {
                for (INT i = 0; i < NX; i++) {
                        printf("%8.4f ", in[IC]);
                }
                printf("\n");
        }
        printf("\n");
}

void print2Display(C_REAL *in, const int start, const int end, const int nrow, const int myRank,
                   const int nProcs)
{
        int nGhostLayers = 2;
        for (int j = 0; j < nrow + nGhostLayers; j++) {
                for (int i = 0; i < NX; i++) {
                        printf("%8.4f ", in[IC]);
                }
                printf("\n");
        }
}

int main(int argc, char **argv)
{
        int nProcs; /* number of processes */
        int myRank; /* process rank */
        int src;    /* handles for communication, source process id */
        int dest;   /* handles for communication, destination process id */
        int start;  /* start index for each partial domain */
        int end;    /* end index for each partial domain */
        int nrow;   /* number of row needed to be allocated by local array */

        MPI_Init(&argc, &argv);                 /* initialize MPI */
        MPI_Comm_size(MPI_COMM_WORLD, &nProcs); /* get the number of processes */

        int nDims = 1; // dimension of Cartesian decomposition 1 => slices
        int dimension[nDims];
        int isPeriodic[nDims];
        int reorder = 1; // allow system to optimize(reorder) the mapping of processes to physical c
ores

        dimension[0]  = nProcs;
        isPeriodic[0] = 0; // periodicty of each dimension

        MPI_Comm comm1D; // define a communicator that would be assigned a new topology
        MPI_Cart_create(MPI_COMM_WORLD, nDims, dimension, isPeriodic, reorder, &comm1D);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank); /* get the rank of a process after REORDERING! */
        MPI_Cart_shift(comm1D, 0, 1, &src,
                       &dest); /* Let MPI find out the rank of processes for source and destination
*/

        // Looking at neighbores for each proces
        printf("myRank=%d mySource=%d myDestination=%d\n", myRank, src, dest);
        MPI_Barrier(MPI_COMM_WORLD);

        // Mesh Decompotistion
        decomposeMesh_1D(NY, nProcs, myRank, &start, &end);
        nrow = (end − start);

        // Initializing Matrix
        int nGhostLayers = 2;
        int AllocSize    = (nrow + nGhostLayers) * NX;
        REAL *local_grab, *local, *local_new;
        local      = (REAL *) calloc(AllocSize, sizeof(*local));
        local_new  = (REAL *) calloc(AllocSize, sizeof(*local_new));
        local_grab = (REAL *) calloc(AllocSize, sizeof(*local_grab));

        REAL *theta     = (REAL *) calloc(NX * NY, sizeof(*theta));
        REAL *theta_new = (REAL *) calloc(NX * NY, sizeof(*theta_new));
        initializeM(theta);

        // Scattering all elements across nProcs
        MPI_Scatter(theta, nrow * NX, MPI_DOUBLE, local_grab, nrow * NX, MPI_DOUBLE, MASTER,
                    MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);

        // Pushing each element in local_grab data back by NX and assigning it to local
        for (int i = 0; i < AllocSize - NX; i++) {
                local[i + NX] = local_grab[i];
        }

        exchange_SendRecv(local, src, dest, nrow, myRank);
        SolveHeatEQ(local, local_new, nrow);

        // Starting the solver
        REAL *tmp;
        for (int iter = 0; iter < MAXITER; iter++) {
                exchange_SendRecv(local, src, dest, nrow, myRank);
                SolveHeatEQ(local, local_new, nrow);
                MPI_Barrier(MPI_COMM_WORLD);

                tmp       = local;
                local     = local_new;
                local_new = tmp;
        }

        MPI_Gather(local + NX, nrow * NX, MPI_DOUBLE, theta_new, nrow * NX, MPI_DOUBLE, MASTER,
                   MPI_COMM_WORLD);

        if (myRank == MASTER) {
                printf("***********FINAL OUTPUT AFTER GATHER***********\n");
                outputMatrix(theta_new);
        }

        // Deallocating Arrays
        free(local);
        free(local_new);
        free(local_grab);
        free(theta);
        free(theta_new);

        local      = NULL;
        local_new  = NULL;
        local_grab = NULL;
        theta      = NULL;
        theta_new  = NULL;

        MPI_Finalize();
        return EXIT_SUCCESS;
}
```

V. Sample Output

```
myRank=0 mySource=-1 myDestination=1
myRank=1 mySource=0 myDestination=-1
***********FINAL OUTPUT AFTER GATHER***********
  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  0.0000   3.3711   5.3698   7.2038   7.7410   7.7410   7.2038   5.3698   3.3711   0.0000
  0.0000   6.1373  12.1563  14.3559  16.0200  16.0200  14.3559  12.1563   6.1373   0.0000
  0.0000  15.6790  21.0797  26.1842  27.3081  27.3081  26.1842  21.0797  15.6790   0.0000
  0.0000  12.5295  24.9797  27.1165  28.8539  28.8539  27.1165  24.9797  12.5295   0.0000
  0.0000 100.0000 100.0000 100.0000 100.0000 100.0000 100.0000 100.0000 100.0000   0.0000
```