COMP605 - Final Project

Akhil Perimbeti

May 19, 2022

Solving the 2D Laplace Equation - Hybrid Implementation

The basic aim of parrallel programming is to decrease the runtime for a solution (optimization), as well as increase the size/domain of the problem that can be solved (scaling up). In this project we will create a hybrid architecture implementation using OpenMP and MPI in order to solve the 2D Laplace equation using an iterative (Jacobi) algorithm. Conventional parallel programming practices involve a a pure OpenMP implementation on a shared memory architecture or a pure MPI imple- mentation on distributed memory computer architectures. The largest and fastest computers today employ **both shared and distributed memory architecture** (hybrid). This gives a flexibility in tuning the parallelism in the programs to generate maximum effciency and balance the computational and communication loads in the program. A wise implementation of hybrid parallel programs utilizing the modern hybrid computer hardware can generate massive speedups in the otherwise pure MPI and pure OpenMP implementations.

OpenMP

OpenMP is an API used for writing multithreaded applications. It takes advantage of implementing and using multiple proccessors on a single memory node. Essentailly, OpenMP is a way to program on shared memory devices. This means that the parallelism occurs where every parallel thread has access to all of your data. The benefits of using a Pure OpenMP implementation include low latency and high bandwidth, allows for implicit communication and load balancing, and finally it is relatively easy to implement. However, OpenMP can be used **only on shared memory machines**, and the threading often occurs in a random order. (Refer to Figure 1). $\longrightarrow$

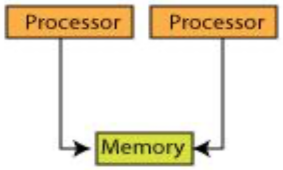

Figure 1: Shared Memory System - Ideal for OpenMP

MPI (Message Passing Interface)

MPI accounts for the fundamental limitation of OpenMP in that MPI is portable to **both distributed and shared memory machines**. It a way to program on distributed memory devices, meaning that the parallelism occurs where every parallel process is working in its own memory space in isolation from the others. The benefits of using a Pure MPI implementation is that it scales to multiple shared memory machines with no data placement problems. However, MPI can often be difficult to implement and debug. It has high latency and low bandwidth. Even if there are multiple processors sharing a memory, MPI implementations will treat each processor as separate, and so explicit communications between them is required, despite the fact that memory is being shared. (Refer to Figure 2) $\longrightarrow$

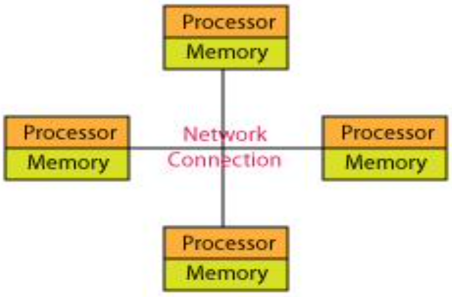

Figure 2: Distributed Memory System - Ideal for MPI

Hybrid Architectures

The theory behind using a OpenMP+MPI Hybrid implementation is to utilize the best from the Pure OpenMP and Pure MPI approaches. The basic concept is to use MPI across the nodes and OpenMP within the node. This allows to avoid the extra communication overhead with MPI within a single node. Hybrid programming may not always be beneficial, since it depends heavily on the problem statement and computational and communication loads. However, if it works on a problem, it can give considerable speedups and better scaling. We will explore the benefits and drawback for this particular type of hybrid architecture. (Refer to Figure 3). $\longrightarrow$

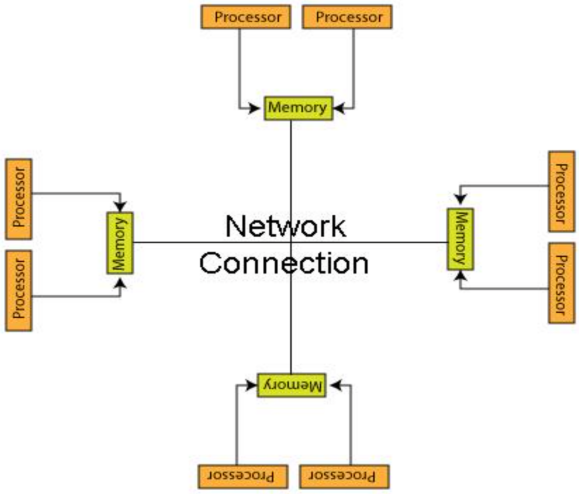

Figure 3: Shared and Distributed memory. Suitable for MPI and/or Hybrid implementations

Potential Advantages of OpenMP + MPI:

- Reducing memory requirements
- Improving performance $\longrightarrow$ Scalability (exploits additional layers of parallelism, reduces communication overheads and load imbalance)

Potential Disadvantages of OpenMP + MPI:

- Development & maintanence costs
- Portability
- Performance pitfalls (idle threads, synchronization, false sharing)
- How many threads? (introduces another tunable parameter that can affect system performance).

Problem Introduction

The general form of the 2D Laplace Equation used to describe the electric potential u (V) is shown as follows:

$$\frac{\delta^2 u}{\delta x^2} + \frac{\delta^2 u}{\delta y^2} = 0 \quad OR \quad u_{xx} + u_{yy} = 0$$

,

The initial values and boundary conditions for this problem are as follows: (BC's and IC's from HW3) $\longrightarrow$ (for a Matrix of size **NxM** split into 2D 'bricks' using a total of 4 processors - Domain Decomposition)

Initial Values and Boundary Conditions Dimensions: [NxM]

$$\begin{bmatrix} -1 & \dots & \dots & -1 & -1 & \dots & \dots & -1 \\ 0 & 0 & \dots & 0 & 1 & \dots & 1 & 1 \\ 0 & \ddots & & \vdots & \vdots & \ddots & & 1 \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ \vdots & & & 0 & 1 & & & 1 \\ 0 & 0 & \dots & 0 & 1 & \dots & 1 & 1 \\ 2 & 2 & \dots & 2 & 3 & & & 3 \\ \vdots & \ddots & & \vdots & \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ 2 & & & 2 & 3 & & & 3 \\ 2 & 2 & \dots & 2 & -3 & \dots & 3 & 3 \\ -1 & \dots & \dots & -1 & -1 & \dots & \dots & -1 \end{bmatrix}$$

where the upper and lower boundaries are set to a value of -1, and the interior points are simply set to the rank of the processes solving that section (Process 0, Process 1, Process 2 or Process 3). My code is built so that it is possible to scale up and increase the size of the matrix, and increase the number of processors and threads as well.

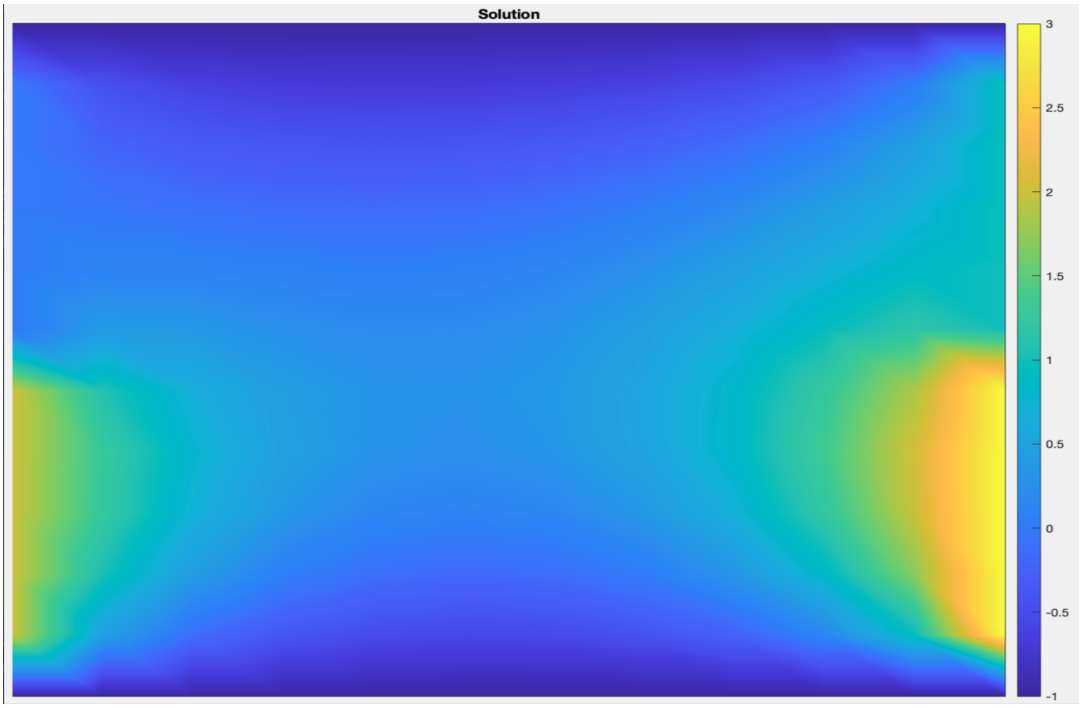

Figure 4: Approximated Solution Plot for the 2D Laplace Equation

Methods

We can approximate the 2^{nd} order derivatives in the solve the 2D Laplace equation by using the **Second-order Central Finite Difference Scheme** in terms of the mesh function $U_{i,j}$ as denoted below:

$$u_{xx} \longrightarrow \quad \frac{\delta^2 u}{\delta x^2} \approx \frac{U_{i,j+1}-2U_{i,j}+U_{i,j-1}}{\Delta x^2}, \qquad u_{yy} \longrightarrow \quad \frac{\delta^2 u}{\delta y^2} \approx \frac{U_{i+1,j}-2U_{i,j}+U_{i-1,j}}{\Delta y^2}$$

Plugging back into the original 2D Laplace Equation: $u_{xx} + u_{yy} = 0$

$$\frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{\Delta x^2} + \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{\Delta y^2} = 0$$

Let $r_x = \frac{1}{\Delta x^2}$ and $r_y = \frac{1}{\Delta y^2}$ and simplify further $\longrightarrow$

$$(r_x + r_y) \cdot (-2U_{i,j}) + r_y \cdot (U_{i,j+1} + U_{i,j-1}) + r_x \cdot (U_{i+1,j} + U_{i-1,j}) = 0$$

We can group all of these terms ($r_x = r_y = \psi$) and divide them over to the RHS of the equation, to cancel out:

$$\psi \cdot [-4U_{i,j} + U_{i,j+1} + U_{i,j-1} + U_{i+1,j} + U_{i-1,j}] = 0 \quad \longrightarrow$$

$$U_{i,j} = \frac{1}{4}\left[U_{i,j+1} + U_{i,j-1} + U_{i+1,j} + U_{i-1,j}\right]$$

Jacobi Iteration Algorithm

1. Given a system of linear equations $\rightarrow Au = b$, where the known $n \times n$ matrix A is symmetric, positive definite, and real, and b (0 for Laplace) is known as well. The solution is denoted by u.
2. An initial guess is made for $u^0_{i,j}$ and is set as a value of either 0 or 1.
3. On the next iteration, the value of $u^1_{i,j}$ is updated using CFD $\rightarrow$ $u^1_{i,j} = \frac{1}{4}[u^0_{i+1,j} + u^0_{i-1,j} + u^0_{i,j+1} + u^0_{i,j-1}]$
4. The algorithm continues to iterate until the stopping condition is satisfied. The stopping condition is triggered when the normalized difference between the calculated values of consecutive iteration updates falls below the tolerance threshold $\rightarrow$ (u^η indicates the solution values at the η^{th} iteration).

$$\sqrt{(u^{\eta}_{i,j} - u^{\eta-1}_{i,j})^2} < TOL \ ,$$

If tolerance is set as 10^{-P}, then the total number of iterations for this method to converge would be $\approx \frac{1}{2} \cdot P \cdot N^2$

The value for the central node $U_{i,j}$ is a calculated as the average of its 4 surrounding neighbors

---

Hybrid OpenMP + MPI Execution Overview

A single MPI process is launced on each SMP node(Symmetric Multi-Processor - shared memory). Each process then spawns 4-32 threads on each one of these SMP nodes. Then, after every parallel OpenMP iteration within each SMP code (executing Jacobi iterative algorithm and convergence checking), the master threads (thread 0) for each SMP node communicate with each other using MPI calls (row and column exchanges). And then again the iteration in OpenMP within each node is carried out using threading until it is complete (the convergence criteria is met). The total execution time is recorded, and the results from each of the SMP nodes (local arrays) are then gathered and sent back to Process 0. The local matrices are iterated through and the calculated interior solution values are selected based on the processor rank, with regards to it position in the final solution output matrix.

Results & Analysis

Table 1 below shows the summary of the implementation of the hybrid architecture scaled to several different grid sizes, utilizing varying numbers of MPI processors and OpenMP thread counts. The "Time elapsed (s)" column in the table below was calculated as the average time in seconds that a single process took to perform the Jacobi iteration completely enough satisfy the convergence criteria. The rows in the table with the $\star$ symbol indicate the parameters (number of processors and number of threads) where the hybrid implementation performed with high efficiency (speed) for those particular grid sizes. It is important to note that initializing too many threads can have a clear detrimental effect on system performance, much more so than too many processors.

N (rows) $\times$ M (cols)	Number of Processors	Number of Threads	Total iterations	Time elapsed (s)
$\star$ 40×20	2	2	$\star$ 141	$\star$ 0.003131
40×20	2	4	141	0.003882
40×20	2	8	141	0.005269
40×20	4	4	270	0.009672
160×80	4	4	2287	0.115675
$\star$ 160×80	4	8	$\star$ 2287	$\star$ 0.114883
160×80	4	16	2287	0.148604
160×80	8	4	3598	0.168122
640×320	4	8	12287	2.12098
$\star$ 640×320	4	32	$\star$ 12287	$\star$ 1.62803
640×320	8	4	29580	4.515193
640×320	8	8	29580	3.567819
2560×1280	4	8	42290	66.660831
$\star$ 2560×1280	4	32	$\star$ 42290	$\star$ 37.88516
2560×1280	8	8	138631	114.63486
2560×1280	8	16	138631	113.67162

Table 1: Hybrid OpenMP+MPI Implementaion $\longrightarrow$ Scalability

Hybrid OpenMP + MPI vs. Pure MPI implementation

Table 2 below displays the results of the performance comparison between the Hybrid architecture and the Pure-MPI implementations. Both of these processes are still performing the 2D domain decomposition to solve the 2D Laplace equation using Jacobi iterations, so the total iterations needed to converge will remain the same for both implementations. So, we are really looking at the differences in execution times. Refer to Figure 5 below.

Number of Processors	Hybrid - Time (s)	PureMPI - Time (s) -	Total iterations
2	14.453040	46.160915	9537
4	18.687208	57.791802	23642
8	29.701562	85.964917	68977
10	28.155446	76.761116	75079
16	27.092428	62.696833	90821
20	21.512803	42.893708	77547
32	13.180990	18.862633	46398
40	8.93591	11.296919	34436
50	6.044687	7.017054	23573

Table 2: Hybrid (threadcnt = 4) vs Pure-MPI Implementaion $\rightarrow$ Fixed grid size: [1280×640]

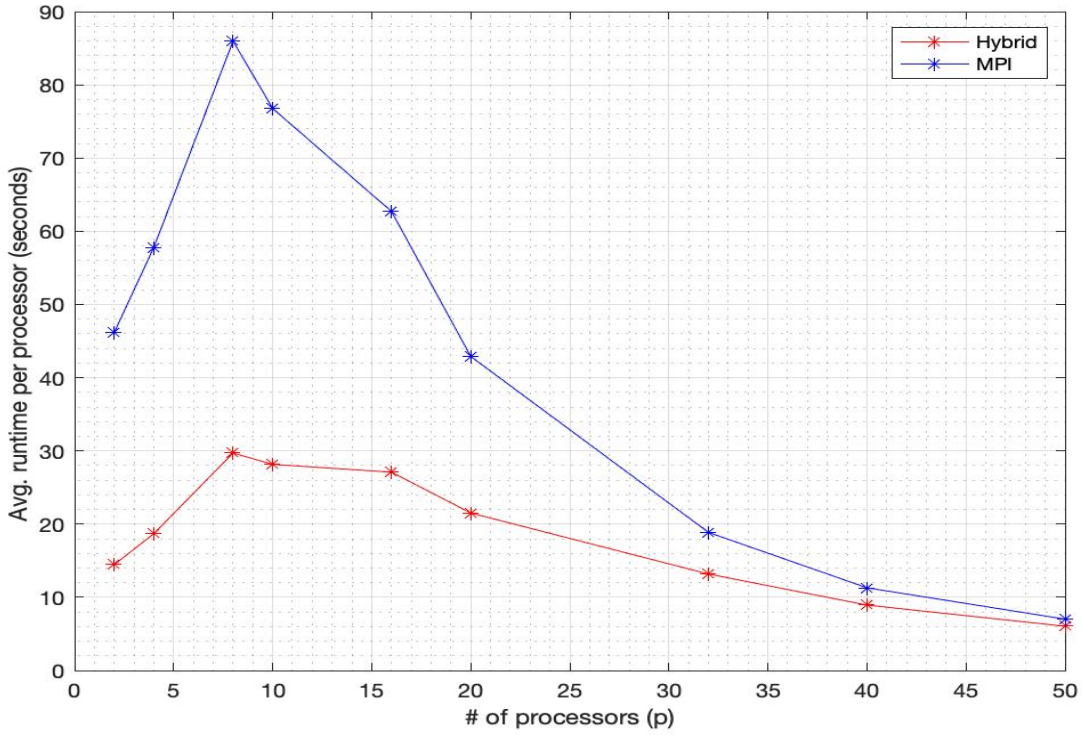

Figure 5: Performance Comparison of Hybrid vs. Pure-MPI approaches for a Fixed Grid Size: $[1280 \times 640]$

The data results from Table 2, plotted on Figure 5, shows that our Hybrid OpenMP+MPI implementation clearly surpasses the Pure-MPI approach, when applied to solve the 2D laplace equation using the Jacobi iterative algorithm with a 2D domain decomposition. The hybrid scheme seems to faster compared pure-MPI scheme for the scope of this problem (and this specific grid size). This might be attributed to the shared memory system of OpenMP, that allows for easier load-balancing between threads compared to MPI processes. However this is not necessarily always beneficial for Hybrid systems. If you look at the plot closely, you can difference is very large when using a fewer number of processors, but closes fast as the number of processors increase. Note how rapidly the change in the speed occurs for the pure MPI system as the number of processors increase compared to that of the hybrid implementation. (NOTE: For this problem case, when the number of processors $p \geq 64$, the speed of the hybrid system changes dramatically - experiencing a slowdown in an almost exponential fashion).

This seems to point towards flaws that exist within Hybrid OpenMP+MPI architectures. A common synchronisation mechanism in OpenMP is the barrier. (implicit barriers are placed at the end of parallel loops and parallel regions). OpenMP does not have high-level point-to-point synchronisation mechanisms, where one thread can wait for another thread to reach a given point in the program. Synchronising threads at barriers also introduces the risk of load imbalance, if all the threads do not reach the barrier at the same time. Adding OpenMP to an MPI code therefore typically introduces a number of these thread barriers, which may be more expensive than point-to-point message passing inside a node. (especially as the number of processors grow larger)

In conclusion, hybrid programming is an attempt to maximize the best from both OpenMP and MPI paradigms. However, outside of only a few select, problem-specific cases, it does not necessarily always imply better performance.

(END)