Utilizing MPI to Identify High-Risk Cardiac Patients

Sarah Parker
Department of Computer Science
University of Tennessee, Chattanooga
Chattanooga, Tennessee, USA
zcz774@mocs.utc.edu
November 20, 2024

***Abstract*—In healthcare, the use of electronic health records has produced pervasive amounts of data on patients. These data points can be used to inform clinicians, provide risk-assessment points for hospitals, and improve patient outcomes. However, due to the volume of information, processing this data can be tedious and time-consuming. This paper shows how using an optimized parallel algorithm for data processing can significantly reduce processing times.**

***Index Terms*—healthcare, parallel algorithm, MPI**

I. Introduction

Chest pain is a common complaint in emergency departments (ED) nationwide. In 2021, they accounted for more than 6.5 million ED visits [1]. However, chest pain complaints are not always related to coronary artery disease (CAD). Some alternative diagnoses related to chest pain include pneumonia, muscle strain, anxiety, and acid reflux disease. To provide appropriate interventions, it is vital that clinicians quickly identify chest pain patients whose complaints are likely related to coronary artery disease. The HEART score is a validated scoring tool used in EDs to risk-stratify patients with chest pain [2], [3]. Based on the results of the HEART score, patients are placed into a low, medium, or high-risk category. These risk categories correlate to a percentage of the likeliness of experiencing an adverse cardiac event in the next six weeks, such as myocardial infarction or death.

A patient's HEART score can be calculated quickly during an ED visit. However, hospitals and analysts may want to find out what percentage of chest pain patients are admitted for further coronary workups vs. those discharged home. Due to the large volume of chest pain patients that come through the ED, evaluating those patients calculated as moderate to high-risk can be time-consuming as it requires careful evaluation of electronic health records (EHR). Therefore, We seek to evaluate the performance efficiency of processing patient data related to HEART scores by using parallel versus sequential algorithms.

II. Literature Review

Parallel computing has been applied extensively in healthcare over the past decade. The development of the graphics processing unit (GPU) in the early 2000s opened the door for exponential performance improvements utilizing parallel computing. Several subspecialties in healthcare rely heavily on parallel computing to analyze data. Interestingly, scientists also use structures from the human body to develop parallel computing techniques [4].

The brain is a highly complex organ that can carry out billions of instructions per minute [4]. This led researchers to design software simulators with the help of GPUs to model the brain and its computational abilities. Neuromorphic hardware is based on the architecture of the nervous system. This hardware utilizes large amounts of parallel arrays [4].

Genomic research and analysis also utilize parallel computing. Dong *et al.* [5] propose a tool named ProteinSPA, which uses mpiBlast to employ parallelism for protein structure prediction. eHive is a program that uses MySQL to process comparative genomic analysis computations in a parallelized form [6]. Morales and Goloboff [7] showed a significant speedup of phylogenetic analysis with the use of a program called TNT, which employs MPI parallel computations. In their review of methods for genomic sequencing, Zou et. al. [3] found significant performance improvement was achieved using various parallel algorithms and techniques.

GPUs can refresh millions of pixels per second, making them an ideal choice in image composition for CT, MRI, and PET scans [8]. In the early 2000s, the ability to perform floating-point operations and programmable features paved the way for the widespread use of GPUs in graphical and non-graphical computations. When applying the NLM filter and algorithm to MRI image processing, Kalaiselvi *et al.* [9] showed a speedup of 338x for the GPU when compared to the CPU.

III. Criteria and Methods

A. HEART Score

There are 5 categories of criteria to evaluate with the HEART score. Clinicians assign points based on each category's results. The category points are then summed to calculate the total score.Table I shows the criteria requirements and associated scoring values. An associated risk value is given based on the total score. Patient scoring is segmented into 0-3 (low-risk), 4-6 (medium-risk), and 7 or higher (high-risk). Low-risk patients have a 1.6% chance of experiencing an adverse cardiac event in the next six weeks, medium-risk

HEART Score									
History		**Age**		**EKG**		**Risk Factors**		**Initial Troponin**	
Slightly suspicious	0	$\leq$ 45 yo	0	Normal	0	No known risk factors	0	$\leq$ normal limit	0
Moderately suspicious	+1	45-64 yo	+1	Non-specific repolarization	+1	1-2 risk factors	+1	1-3x normal limit	+1
Highly suspicious	+2	$\geq$ 65 yo	+2	Significant ST deviation	+2	$\geq$ 3 risk factors	+2	$\geq$ 3x normal limit	+2

TABLE I
HEART SCORE POINTS

patients have a 13% chance, and high-risk patients have a 50% chance [2]. This information is helpful to clinicians who want to discharge a low-risk patient [10]. For example, a chest pain patient whose symptoms are likely related to acid reflux and also receives a total HEART score of one can be safely discharged home with appropriate follow-up. On the other hand, it is also beneficial to clinicians who wish to admit a patient for further workup by providing supportive documentation for their justification.

B. Code and Data Generation

EHRs contain significant amounts of information that must be parsed to identify specific calculation components. This program was written to mimic some(but certainly not all) of the values that would be found in an EHR. The first step was to create a pseudo-database of patient information. This information was generated randomly within the program. It included patient identification numbers, systolic and diastolic blood pressure measurements, heart rate, oxygen level, lab values, and all the criteria associated with the HEART scores. This ensured the program used only the required data to calculate the HEART score and still saved all of the patient's associated data in the generated output file. One million patients were generated to ensure a sufficient sample size for both programs.

Then, a function was created to sum the total points calculated for each patient's HEART score. They were subsequently assigned to the low, medium, or high-risk category. For this paper, we were only interested in high-risk patients (those likely admitted to the hospital for further evaluation). These patients were placed in a comma-separated values (CSV) file with all their associated data so as to be comparable to saving a full EHR record. This way, the data can be accessed later if more information is needed. Lastly, the number of high-risk patients was totaled and taken as a percentage of the total population. This percentage gives an accurate representation of patients with true coronary disease who come through the ED with a complaint of chest pain.

Two versions of the program were written in C++. The first version created and analyzed the data using a standard sequential algorithm. The second code was created, and the data was analyzed using the Message Passing Interface (MPI) to take advantage of parallelization techniques. MPI was chosen as it allows multiple processes to run simultaneously, thereby improving performance efficiency [11]. Both programs were executed on a high-performance computing (HPC) node at the University of Tennessee, Chattanooga (UTC), to fully utilize the advantages of MPI and its ability to run on multiple cores.

C. Sequential

The sequential code was written using a standard iteration. Each iteration initially assigned values to all patient record data points. This created our pseudo EHR. Then, within each iteration, a for-loop determined a patient's risk category and the subsequent operations. If patient was assigned to the high-risk category, the high-risk counter was incremented, and the patient's data was saved into a vector. However, if the patient's risk were moderate or low, the for-loop would exit and return to the next iteration. The for-loop is executed 1,000,000 times in a sequential fashion to generate patient data and then analyze each patient's risk category. A timing function was added to assess the execution time of the program. The timer began when the iteration started and ended when all patient data had been generated and analyzed. The high-risk patient data was then saved as a CSV file. Finally, on the command line, the total number of high-risk patients and the percentage of high-risk patients to the total number of patients were outputted for a quick review of the results.

D. MPI

The MPI environment was initially set using the MPI_Init, MPI_Comm_rank, and MPI_Comm_size. MPI_Init is the root command to enable communications within MPI. The function must appear once in order to start the MPI library. MPI_Comm_rank assigns each process an individual ID number. This is used to assign specific processes computations and data, when workload is being distributed. MPI_Comm_size is used to specify the total number of processes. This function takes a parameter called &size which is a pointer to the number of processes. The number of processes can be specified on the command line during execution.

Patient data was then generated and assigned to a specific process using a vector. A non-blocking technique MPI_Gather was used to gather all the generated data. This technique was used as it is thread-safe and does not require blocking mechanisms to prevent race conditions. Each process was assigned a specific number of patients to balance the workload and improve efficiency. The size of the data (number of patients) was calculated for each process and then added together to get the total number of patients. This number was used to allocate the appropriate size for the globalPatientData variable. High-risk patients were selected from each process using MPI_Reduce by calculating the sum of the scores and selecting patients with a score of > 7. This method effectively reduces the number in each process to only contain the high-risk patients. The data from the vector is flattened into a string array named flatRecvCounts. The array allows MPI to process the data more efficiently using MPI_Gatherv. MPI_Gatherv differs from MPI_Gather in allowing different data sizes from each process. This is important as there is likely a different amount of high-risk patients in each process. The high-risk

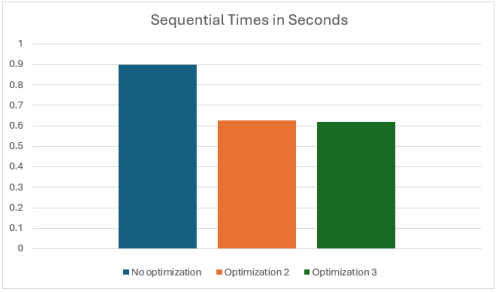

Fig. 1. Sequential Performance Times

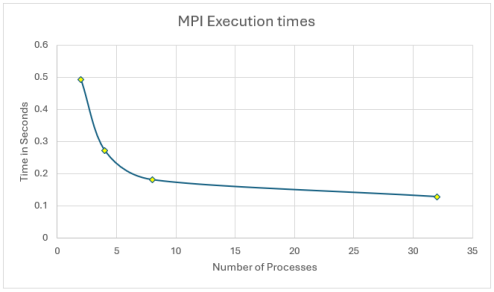

Fig. 2. MPI Performance Times

patients were then gathered back together into the root process. After the data is gathered and returned to the root process, a CSV file containing all the patient data from the high-risk population is generated. A timing function was added using MPI_Wtime() to calculate the total program execution time.

IV. Results

A. Sequential

The sequential version of the code was compiled in three different ways. The first compilation was without any optimizations using the command:

```
g++ -Wall <file name> -o <program name>
```

The second compilation specified that the compiler should perform moderate optimizations to improve performance. It was written using this command:

```
g++ -Wall -O2 <file name> -o <program name>
```

Finally, the code was compiled using the maximum compiler optimizations. This was written using the command

```
g++ -Wall -O3 <file name> -o <program name>
```

. The first program was executed and completed in 0.90s. This second program was executed and completed in 0.63s. The third program was executed and completed in 0.62s. As seen, the was a neglibigle amount of improvement between the second and third level of compiler optimizations. The comparison of these times can be seen in Fig. 1.

B. MPI

The MPI version of the code was tested in four different ways. Since MPI utilizes parallelization through multiple processes, compiler optimizations were not used. The same command was used to compile all four programs,

```
mpic++ <file name> -o <program name>
```

When the program was executed, the number of processes was explicitly specified using the command:

```
mpirun -np <number of processes> <file name> -o <program name>
```

The number of processes was increased with each execution, and the workload was distributed in a parallel function. The first version used two processes, and the resulting time was .49s. The second version of the code used four processes, and the execution time was 0.27s. The third version of the code used eight processes and resulted in an execution time of 0.18s. Lastly, the code utilized 32 processes, resulting in an execution time of 0.13s. Fig. 2 shows a graph of the times and processes used.

C. Speedup

Speedup is a calculation that evaluates program efficiency by using the execution times of a program. It is calculated by dividing the original time by the new time. A speedup of > 1 is expected for a program to be more efficient. The speedup for the sequential version of the code was calculated by taking the execution time without optimization and dividing it by the optimized level 3 version. The speedup achieved was 1.4x faster. When evaluating the MPI version of the code, the speedup achieved when increasing the processors from 2 to 32 was 3.8x faster. Finally, when comparing the fastest sequential version to the fastest MPI version, the total speedup was 4.8x. Fig. 3 shows the speedup values.

The fastest optimized sequential time was 0.61s, slightly faster than the parallel program using two processes at 0.49s. This equates to a speedup of 1.2x. However, when the fastest optimized execution is compared to the parallel version using 32 processes, a more significant speedup is achieved. This demonstrates that using a parallel algorithm for data processing in this code was more efficient than using a non-parallel algorithm.

Therefore, the limiting factor in the parallelized version is the number of processes available. Given that a standard computer typically has 2-4 cores available, there would be little difference between the two program versions. Therefore, it is recommended that a computer with at least 8 available cores be used to see a significant performance improvement using the parallel algorithm.

V. Conclusion

The HEART score is an effective tool for determining the risk of cardiac events in the next 30 days. Evaluating data from these scores can help clinicians determine treatment plans for

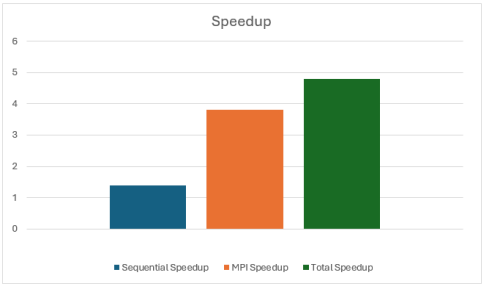

Fig. 3. Speedup

patients with chest pain. Hospitals can also utilize the data to evaluate the number of high-risk patients who were discharged and need follow-up vs. those who were admitted for further workups.

The volume of patient data collected can make evaluating individual HEART scores tedious. We have evaluated the execution time differences between sequential and parallel algorithms. Through this approach, we demonstrated significant speedup when using a parallel algorithm with MPI. The limitations on the parallel version can be eliminated by ensuring the machine executing the code has the necessary minimum recommended number of cores.

Hospitals and clinicians can use the data from this program to evaluate patients further and improve outcomes for those with chest pain.

REFERENCES

[1] "Women and Black adults waited longer in ER for chest pain evaluation," American Heart Association. https://newsroom.heart.org/news/women-and-black-adults-waited-longer-in-er-for-chest-pain-evaluation

[2] "HEART Score," www.heartscore.nl. https://www.heartscore.nl

[3] Y. Zou, Y. Zhu, Y. Li, F.-X. Wu, and J. Wang, "Parallel computing for genome sequence processing," *Briefings in Bioinformatics*, vol. 22, no. 5, Art. no. bbab070, Sep. 2021. doi: 10.1093/bib/bbab070.

[4] L. A. Pastur-Romay, A. B. Porto-Pazos, F. Cedron, and A. Pazos, "Parallel Computing for Brain Simulation," *Current Topics in Medicinal Chemistry*, vol. 17, no. 14, pp. 1646–1668, 2017. doi: 10.2174/1568026617666161104105725.

[5] S. Dong, P. Liu, Y. Cao, and Z. Du, "Grid Computing Methodology for Protein Structure Prediction and Analysis," in *Parallel and Distributed Processing and Applications – ISPA 2005 Workshops*, Berlin, Springer, 2005, pp. 257–266.

[6] K. Ocaña and D. de Oliveira, "Parallel Computing in Genomic Research: Advances and Applications," *Advances and Applications in Bioinformatics and Chemistry*, vol. 8, pp. 23–35, 2015. doi: 10.2147/AABC.S64482.

[7] M. E. Morales and P. A. Goloboff, "New TNT Routines for Parallel Computing with MPI," *Molecular Phylogenetics and Evolution*, vol. 178, 107643, 2023. doi: 10.1016/j.ympev.2022.107643.

[8] "Graphics Processing Unit," *Wikipedia*, Nov. 2, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Graphics_processing_unit.

[9] T. Kalaiselvi, P. Sriramakrishnan, and K. Somasundaram, "Performance of Medical Image Processing Algorithms Implemented in CUDA Running on GPU-based Machine," *International Journal of Intelligent Systems and Applications*, vol. 10, no. 1, pp. 58–68, 2018. doi: 10.5815/ijisa.2018.01.07.

[10] W. Brady and K. de Souza, "The HEART score: A guide to its application in the emergency department," Turkish Journal of Emergency Medicine, vol. 18, no. 2, pp. 47–51, Jun. 2018, doi: https://doi.org/10.1016/j.tjem.2018.04.004.

[11] "Message Passing Interface," Wikipedia, Jul. 26, 2021. https://en.wikipedia.org/wiki/Message_Passing_Interface