Digital Object Identifier

Using Markov Chains to Predict Outcomes in Baseball

JESSE GAILBREATH[1], JANSEN LONG[2], RYAN MORSE[3], DILAFRUZ SHAMSIEVA[4], ELIJAH WHITE[5]

ABSTRACT

MARKOV Chains have been shown to be an effective way in which baseball teams can statistically select batter/pitcher match-ups that are favorable to them. These calculations could take some time using normal computing methods due to the amount of data that may be useful. Therefore, we will implement an algorithm that solves Markov chains with Major League Baseball statistics and will use parallel processing concepts and technologies in order to create an efficient system for predicting what matchups will be favorable for a given player

I. INTRODUCTION

BASEBALL has a well-defined and clean structure, which allows us to predict the probability of certain outcomes. Markov chain models have been used in advanced baseball analysis since the 1960s. Research has shown that Markov chain models can be used to evaluate runs created for the evaluation of individual and team performances.Using the Markov chain model in our study, we will get a reasonable approximation of the probability of events occurring, from which we will develop our analysis. [1]

In order to get as accurate a prediction as possible, there will need to be a large amount of data taken into account. Many calculations can go into just the prediction of one plate appearance. This is where using parallel processing methods will help in this model. Using parallel processing, we will be able to speed up the prediction of events for use in real time scenarios. This allows us to predict a plate appearance as it occurs.

The data that will be used in this project is from "baseball-savant.com" [3]. This is a website that tracks and stores data from Major League games for public use.

II. LITERATURE REVIEW

FOR real-time baseball prediction, many methods have been proposed and implemented. In baseball, consistently winning the matchup between pitcher and batter is integral to winning games. Markov chains models have been used in sports analysis for years, with applications being implemented for predicting the NCAA Men's Basketball Tournament (Paul and Sokol, 2006), making the decision of when to pull a goalie in a hockey game (Zaman, 2001), and ranking college football teams (Kolbush and Sokol, 2017), among many other examples [3]. The predictive power of Markov Chains can give us a ton of insight into pitcher batter matchup and can even help to predict the outcomes in those matchups. There are numerous scientific papers that impressively demonstrate the effectiveness of Markov Chain.

One of the papers explains a statistical analysis of the stochastic event-transition matrices. The central consideration is a balance between accuracy and (possible) changes in baserunner advancement from season to season. This analysis provides a lower bound to the probabilities that transitions from any baseball state to all others are simultaneously within specified distances. It also highlights additional considerations that must be made for event-based matrices, compared to the (total) transition matrix of the (standard) Markov model [5].

An article done by Beaudoin, David from the journal "Journal of Quantitative Analysis in Sports" develops a simulator for matches in Major League Baseball. Aspects of the approach that are studied include the introduction of baserunning probabilities which were obtained through a large data set, and the simulation of nine possible outcomes for each at-bat [6].

III. SYSTEM DESIGN

THE Markov Chain approach to predicting baseball outcomes has several variations. Our team believes that we have found an approach to the problem that would greatly benefit from the use of parallel processing. We will start with a 19x19 transition matrix representing probabilities of a given player to cause certain outcomes.

Once the data is loaded into the matrix, we then multiply that matrix by itself until the steady state matrix is reached. This is where we see the first possible implementation of parallel processing in this approach. By splitting the 2 matrices into chunks we can parallelize their multiplication. We predict that the computation time will benefit from this process. The steady state will be reached when all the elements to the left of the 1B (Single) column are zero. You are then left with the individual players general probabilities of hitting a single, double, triple, or home run respectively, as well as their batting average, ball, and strikeout percentages. The result is a nicely compressed 12x7 stochastic probabilities matrix which can be calculated for any player with the requisite statistics.

	1B	2B	3B	HR	BIP	BB	K
0-0	**0.110**	**0.047**	**0.003**	**0.079**	**0.379**	**0.171**	**0.210**
0-1	0.094	0.034	0.004	0.063	0.391	0.121	0.293
1-0	0.106	0.053	0.000	0.085	0.344	0.248	0.163
0-2	0.080	0.024	0.000	0.042	0.303	0.096	0.454
1-1	0.103	0.053	0.000	0.067	0.337	0.179	0.262
2-0	0.101	0.053	0.000	0.070	0.254	0.408	0.113
1-2	0.067	0.033	0.000	0.057	0.269	0.130	0.445
2-1	0.084	0.066	0.000	0.086	0.269	0.307	0.188
3-0	0.022	0.037	0.000	0.033	0.106	0.732	0.070
2-2	0.070	0.052	0.000	0.073	0.265	0.207	0.333
3-1	0.037	0.061	0.000	0.054	0.175	0.557	0.115
3-2	0.056	0.056	0.000	0.040	0.177	0.403	0.266

FIGURE 1. Example of Final Steady State Matrix.

From here there are several directions we could go in terms of application for this model. The one that we felt would best benefit from parallel processing is batter-pitcher matchups. To do so is actually quite simple. Once the steady-state matrices are both calculated, to simulate the matchup one only needs to take the average of the 2 matrices. To do so for an entire batter-pitcher lineup outlook would be very computationally taxing, and it will be necessary to apply parallel processing to the algorithm.

IV. DATA COLLECTION AND PREPARATION

THE dataset is collected from the baseball savant website from regular seasons 2017 through September 2022 years. Excel was used to collect, pre-process and store the data. The Python library Pybaseball is used to retrieve data and format it into a Comma Separated File (CSV) to perform the calculation in C/C++. For transition matrix we are using the data of a baseball pitcher Madison Bumgarner and baseball second baseman Jose Altuve changing it into the matrices format.

V. PROJECT BREAKDOWN

MARKOV chain has advantages in terms of speed and precision.From successional data, a Markov model is relatively simple to construct. The primary parameters of dynamic change (in context of the sport's dynamics as it generates massive amounts of data) are summarized in the transition matrix.It provides a comprehensive view of the sports system's evolution over time. It can easily model additional data and new parameters as data collection is improved.

The Markov-Chain approach is generally applied to baseball by breaking a half-inning of play up into 24 different states and attempts to predict the probability of the corresponding state transitions. [7].

While this method is tried and true, our team happened across another use of this method that we found interesting and decided it could benefit from parallelization. In this method we will use a 19x19 Absorbing Markov-Chain to generate stochastic transition matrices for individual players. These probabilities will be calculated based on the Count (Balls-Strikes), and the outcomes of At Bats [3].

In a plate appearance, the batter finds themselves in 1 of 25 different situations (or states). If relevant stats are available, then a transition matrix can be generated for any such player.

$$P = \begin{bmatrix} A_0 & B_0 & C_0 & D_0 \\ 0 & A_1 & B_1 & E_1 \\ 0 & 0 & A_2 & F_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

{A, B, C /8x8 Matrices}
{D, E, F/ 8x1 Column Vectors}
{0's in middle 2 rows are 8x8
Last row contains 24 0's and a 1

25x25

Subscripts are number of outs of plate appearance

FIGURE 2. Example of Final Steady State Matrix.

The 8x8 blocks represent transitions from the current state to runner on first, runner on second, runner on third, runners on first and second, runners on first and third, runners on second and third, and bases loaded respectively. A blocks represent events that do not increase the number of outs. B blocks represent events that increase the number of outs by one, without reaching three. C blocks events that increase the number of outs from 0 to 2. Column D, E and F represent events that increase the number of outs to 3.

1) Breakdown of block A_0

Each of the rows and columns represent a different configuration of runners on bases. The block A0 specifically represents the probabilities of events that start with no outs and do not incur outs.

The matrix in Figure 3 tracks probabilities of runner movements based on the action of the current hitter. Example: The cell in the 6th row and 8th column is accross from R13 and down from RL. That means it represents the probability of going from a runner on first and third to bases loaded without incurring an out. This would most likely be caused by hitting a single.

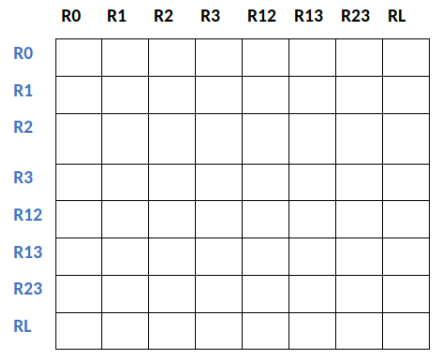


FIGURE 3. Probability of runner movements

These stats can be calculated from relevant stats. Our next job is to find the discrete rules governing the matrix formation. For instance using A_0 as another example. A batter stepping to plate with no runners on base will have no way to go from their current state to a runner on first and third. Thus certain cells can be immediately initialized to 0.

To use the matrix a row vector u_0 will contain 25 values and will represent a simulated current state. To place in linear linear algebra terms P is the transition matrix, u_0 is a state matrix, and a simulated base appearance or "At Bat" can be represented by $u_0 x P_{n+1}$ where P_{n+1} is the transition matrix for the next batter.

VI. MATRIX MULTIPLICATION

MATRICES are widely used since the matrix language is an integral part of the spheres of life. Matrix algebra applies to solving a wide range of important problems because it simplifies the calculation procedure and makes it easier to understand the process. If in mathematics and physics they are used as a compact notation, then in biology they are used in solving real problems of genetics, population, and systematics. In economics, matrices allow, with minimal effort and time, to process a huge and very diverse statistical material (a feature of the socio-economic complex, initial data characterizing the level and structure), as well as to conduct calculations with them. All this information and examples lead to the fact that matrices have been used and will be used in the future. Therefore, we conclude that matrix algebra has been widely used and is still being used, and it will always be relevant in various areas of life.

Matrix multiplication is an essential mathematical operation with many applications in all areas of computer science. To name a few: compactly notating systems of linear equations, linear regression, dimensional reduction. In our research we have decided to use Naïve Matrix Multiplication Algorithm. [8] The algorithm loops through all entries of S and P, and the outermost loop fills the resultant matrix Q.

The naive matrix multiplication algorithm contains three nested loops. For each iteration of the outer loop, the total number of the runs in the inner loops would be equivalent to the length of the matrix. Here, integer operations take O(1) time. In general, if the length of the matrix is N, the total time complexity would be O(N * N * N) = O(N^3).

Algorithm 1: The Naive Matrix Multiplication Algorithm

```
Data: S[A][B], P[G][H]
Result: Q[][]
if B == G then
    for m = 0; m < A; m++ do
        for r = 0; r < H; r++ do
            Q[m][r] = 0;
            for k = 0; k < G; k++ do
                Q[m][r] += S[m][k] * P[k][r];
            end
        end
    end
end
```

FIGURE 4. Pseudocode of the Naïve Matrix Multiplication Algorithm

For our approach we chose to use an absorbing Markov chain. The matrix will begin in a transient state and run through multiple iterations of multiplication until reaching an absorbing or "steady" state. Transition matrices of this type take the form shown in the figure 5 depicting an (n+a)x(n+a) matrix.

R: nxn matrix with transition probabilities between each state

A: nxa matrix giving transition probabilities from each transient state to an absorbing state

N: axn zero matrix

I: axa identity matrix

$$\begin{bmatrix} R & A \\ N & I \end{bmatrix}$$

Figure 3.2: A Generic Transition Matrix for an Absorbing Markov Chain

We're going to use some block matrix notation. Basically, all that means is a particular variable will represent a section of a matrix rather than just an individual element of the matrix. The structure that we have is critical for our model to work. We start with the top left corner. Our is an end-to-end matrix with transition probabilities between each state.

A general probability of moving from one state to another at the end by a matrix section. The right is giving transition probabilities from transient states to absorbing states, and they don't change essentially no matter how many times you multiply through. Due to the structured nature of baseball an individual plate appearance has a relatively small number of possible ways to affect the current state of the game. This can be depicted using an absorbing Markov chain model where the states of the chain are represented by the count, and the possible outcomes of the plate appearance.

In this model there are 19 unique states. The first 12 are transient states, which represent the maximum 12 potential counts a player can play during an at-bat. The other 7 states represent the most common outcomes of a plate appearance.

The graphical representation in Figure 5 of our Markov chain. We are starting out here at the top with the zero account. Batters are stepping up to plate. They have nothing on the field. Nobody's on the field. Nothing is going on. The only way that the state of that inning is going to change is if

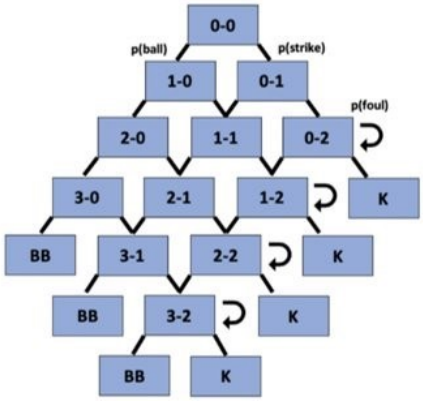


FIGURE 5. Markov Chain for a Generic At-Bat

they either get a ball or a strike. If they hit and they go on base, the next batter steps up and we have a Markov Chain to calculate. We have a total 12 number of counts that the baseball player can be in representing those transient states. And the seven outcomes we can see from those transient states. The data that we are processing, we are pulling this from the MLB Stat cast system. All of this comes down pre-compiled CSV file so we can see over the course of an entire career or even just a particular season, the number of events that a particular player has seen and the types of events, those are the outcomes of events.

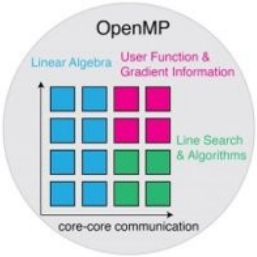
In the actual structure of what we'll be using, we have N by N portion on the top left, which represents our transient states from different counts. We have the N by A portion, which represents the absorbing states that states that essentially represent the outcomes. [3]

The conventional way of calculating is multiplied it by itself several times to reach that transient state or that absorbing state, rather than multiplying through just a single row vector. This simple thing is a lot in terms of global balancing and parallel processing. We start with our normal matrix and once we go through the matrix multiplication process where we multiply this matrix by itself over and repeatedly, we end up with the stochastic, steady state matrix. And this portion is what we would be using for our analysis. We could do a side-by-side comparison with another player, if we wanted to calculate a better picture of match outs. The total time to took to run was less than a second. But if you're calculating this for numerous players, that can get computationally intensive quickly. This is where we're bringing in the matrix multiplication part of this and where we're introducing the parallel processing as well.

VII. PARALLEL PROCESSING APPROACHES

To apply the benefits of parallel processing to our baseball model we will use 2 approaches, a shared memory model with threads, and a distributed memory model.

OpenMP: An API for writing multithreaded Applications. It is a set of compiler directives and library routines for parallel application programmers. Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++. Standardized last years of SMP practice. [9]



OpenMP implements computations using multithreading, in which a "master" thread creates a set of subordinate (slave) threads and a task is computed between them. It is assumed that threads are discovered on machines with multiple processors (the number of processors should not be greater than or equal to the number of threads).

Tasks used by threads in parallel, as well as the data required to perform these tasks, are approached using a special preprocessor directive in the corresponding language - a pragma. For example, a section of Fortran code that must be implemented by multiple threads, each time N of which needs to be implemented.

The number of threads to be started can be regulated both in the most common way by calling library procedures, and from the outside, by using the implementation of the environment.

The key benefits of OpenMP are construction for creating threads (parallel directive), constructs for distributing work between threads, construction for managing work with data (expressions of general and private definitions of a storage class), constructs for streams (critical, atomic and barrier directives), library runtime support routines, environment variables.

The general workflow of OpenMP follows the fork-join model. For this, there can be any number of forking new threads and joining them (even within a parallel region). Shown in Figure 6.

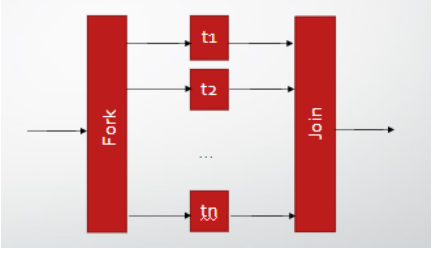


FIGURE 6. OpenMP Workflow

The structure in Figure 7 initiates the parallel section. And inside of the parallel section, we can have our work sharing constructs, we can have our synchronization constructs. We can use run time like library calls, all OpenMP directives.

Then we join the threads, disband and we can continue zero code. Also we could even do another pragma with OpenMP section or we could even parallelize it even more.

```
#include <omp.h>

main () {
int var1, var2, var3;
Serial code
    .
    .
    .
Beginning of parallel section. Fork a team of threads.
Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)
{
Parallel section executed by all threads
Other OpenMP directives
Run-time Library calls
All threads join master thread and disband
}
Resume serial code
    .
    .
    .
}
```

FIGURE 7. OpenMP Program Structure

We have created a standard implementation statement (Figure 8) where we have got pragma omp parallel with standard matrix multiplication algorithm. We haven't reduced the number of operations. The work is speeding up by performing those operations in parallel over multiple threads. We had a flag for steady state to make sure that we broke out of that. Since we needed to put some synchronization into this code, we met some challenges with using double for loops in the code. Since it slows down the work.

```
do
{
STEADY=1;
#pragma omp parallel for private(i,j,k,accm) shared(M1,M2,M3,STEADY)
for (i=0 ; i<ROWS ; i++)
{
        for (j=0 ; j<COLS ; j++)
        {
                accm = 0;
                for (k=0 ; k<ROWS ; k++)
                {       accm += M1[i][k] * M2[k][j];    }

                M3[i][j] = accm;
                if (j<12 && STEADY>0 && accm>0)
                {
                        #pragma omp atomic write
                        STEADY=0;
                }
        }
}

if (!STEADY)
{
        swp = M3;
        M3 = M1;
        M1 = swp;
}
} while(!STEADY);
```

FIGURE 8. OpenMP Implementation

We ended up getting a speed up of 0.01. And then we get speed up all the way to running with four threads, down to 0.0075, and we started losing those benefits at eight threads (0.0076). In Figure 9 you can see the results, red is a serial code running and that's all the same, and OpenMP we got a little faster.

MPI (Message Passing Interface). MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process. A message-passing library specification:

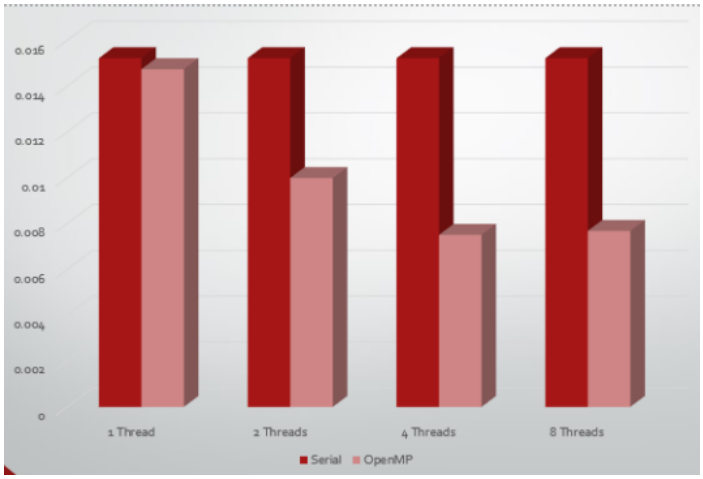
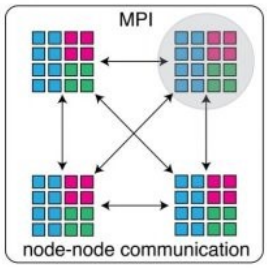


FIGURE 9. OpenMP Results

- extended message-passing model
- not a language or compiler specification
- not a specific implementation or product

MPI is designed to provide access to advanced parallel hardware for end users, library writers, and tool developers. It utilizes separate processes with individual stack and heap. It has point-to-point communication through processes, not pointers.



MPI in general is a flexible and portable way to implement parallel processing. It allows for your heterogeneous systems to communicate. If you are dealing with different types of structures, meaning you may not have the exact same memory structure for one machine to another, MPI standardizes that through the process and allows the communication to go through pretty seamlessly.

It can optimize available performance of a network's communication speed. Outside of just your basic message passing using the single program, multiple data model. Executing through in parallel functions to utilize your hardware in a way that's going to increase your performance dramatically in a lot of cases. In Figure 10 you can see the basic structure for a general MPI program.

The structure of MPI (Figure 10) shows that you have inclusion of headers, non-coordinated running models. The Serial section is at the top. Parallel code is in initialization

for API. After termination of the environment, you need to wrap it up with any serial stuff at the end of that before you return.

Section in the code that we used where the main process was using send and receive. In doing so, we separated the matrix into set size of rows that were passed through to each process. And in doing so, based on the structure you strip down the amount of work done by the number of processes that you were able to utilize. You send it out and then you wait to get it back as it goes through the process. One of the things that is important about doing this is that order is essential. You are pointing to the right.

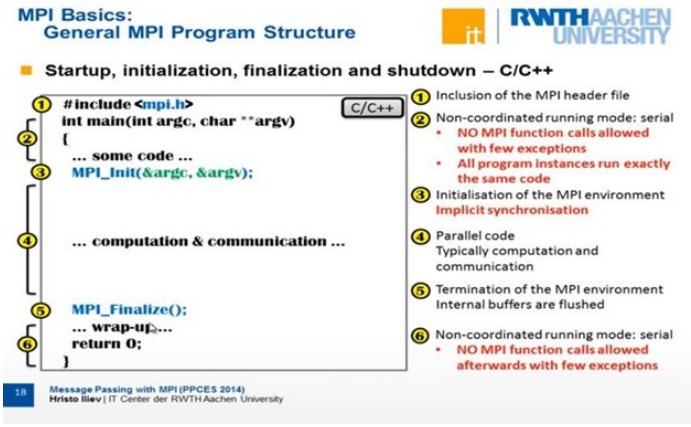


FIGURE 10. MPI Structure

Data movement and worker processes are outside of the main part of the structure. This is where each one of these workers is going to receive that data and start making the computations on that before it starts sending it back to the main to compile all the results and put it into a single location. The interesting thing about MPI is that if you go back and look at the code sections, you'll realize that we're sending back a different address section, a static area that is separate from what was sent in. You are sending in specific pieces of that data when it's making those computations and doing system right that is important that they stay separated. You'll notice that whenever you're doing send and receive, the buffers that are used for your sends and the receives are different.

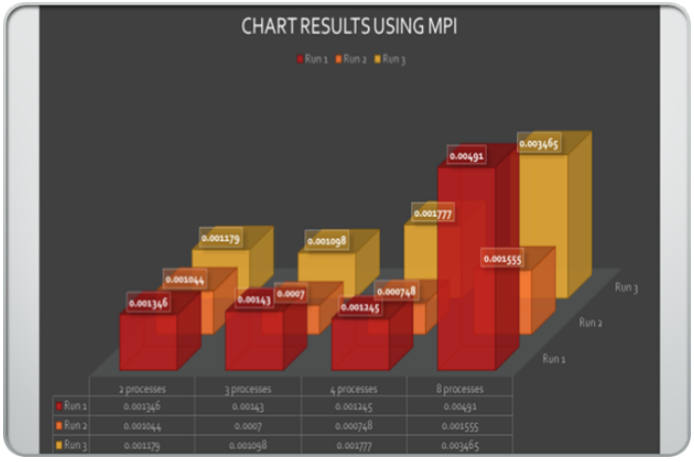
If you look at Figure 11 you can see the MPI processes in action. There are some print statements that are scattered throughout. We can see that the file was opened successfully. We ran two separate files. We ran it with six processes. Once the division was done, there were four extra rows that needed to be split back out and that averaged three rows per chunk. And you can see that as each one was completed, the cycle was completed, and then the new matrix was updated. And you can see as it goes through there, three different cycles processing and you can see that the each one where it says ID are completed a chunk. You can look there and notice that they're not in sequential order all the time, because you may have one of those processes completing a section before

```
(base) jovyan@jupyter-jgg2x:~/MPI$ mpirun -np 6 ./mult
File opened successfully.
File opened successfully.
Number of tasks available= 6
Number of workers that will be assigned= 5
Extra = 4
AVG = 3
DEST= 1 ---EXTRA= 4 ---CFFSET= 0 ---CHUNK = 4
DEST= 2 ---EXTRA= 4 ---CFFSET= 4 ---CHUNK = 4
DEST= 3 ---EXTRA= 4 ---CFFSET= 8 ---CHUNK = 4
DEST= 4 ---EXTRA= 4 ---CFFSET= 12 ---CHUNK = 4
DEST= 5 ---EXTRA= 4 ---CFFSET= 16 ---CHUNK = 3
ID 1 completed a chunk.
ID 3 completed a chunk.
ID 4 completed a chunk.
ID 5 completed a chunk.
ID 2 completed a chunk.
Cycles: 1
M1 updated.
DEST= 1 ---EXTRA= 4 ---CFFSET= 0 ---CHUNK = 4
DEST= 2 ---EXTRA= 4 ---CFFSET= 4 ---CHUNK = 4
DEST= 3 ---EXTRA= 4 ---CFFSET= 8 ---CHUNK = 4
DEST= 4 ---EXTRA= 4 ---CFFSET= 12 ---CHUNK = 4
DEST= 5 ---EXTRA= 4 ---CFFSET= 16 ---CHUNK = 3
ID 1 completed a chunk.
ID 3 completed a chunk.
ID 4 completed a chunk.
ID 2 completed a chunk.
ID 5 completed a chunk.
Cycles: 2
M1 updated.
DEST= 1 ---EXTRA= 4 ---CFFSET= 0 ---CHUNK = 4
DEST= 2 ---EXTRA= 4 ---CFFSET= 4 ---CHUNK = 4
DEST= 3 ---EXTRA= 4 ---CFFSET= 8 ---CHUNK = 4
DEST= 4 ---EXTRA= 4 ---CFFSET= 12 ---CHUNK = 4
ID 1 completed a chunk.
DEST= 5 ---EXTRA= 4 ---CFFSET= 16 ---CHUNK = 3
ID 3 completed a chunk.
ID 4 completed a chunk.
ID 2 completed a chunk.
ID 5 completed a chunk.
Cycles: 3
M1 updated.
```

FIGURE 11. MPI processes in action.

another one. It's not always sequential.

MPI can be incredibly fast. There are multiple ways to use MPI functions to achieve the same result. It Aalows a program to take advantage of hardware and network architecture. Our results had MPI outperforming OpenMP by margins as high as 10x faster.MPI has a very large library, over 125 functions. It is fairly basic to use, parallel program can be written using 6 general functions.MPI can implement user defined error handling routines. It is portable since it is operating at very low level.



One of the things that we found in doing this is that we had a couple of runs that MPI outperformed OpenMP by a rate, it was almost ten times faster. From the MPI results in Figure 12 you'll notice that when we're on three separate sets of runs, different numbers of processes, the results on some of these are impressive compared to OpenMP for the same file, same program.

As far as the process goes, the application used the same algorithm to produce the results. We got some good and fast results. With a general need for speed, whether you use an OpenMP or MPI, there are a lot of different ways you can do parallel programing. As time moves on and as the world becomes more data dependent, being able to access and process this data at a much more effective speed is important.

REFERENCES

[1] S. Deb, "Explore Markov Chains With Examples — Markov Chains With Python," Edureka, May 12, 2020. https://medium.com/edureka/introduction-to-markov-chains-c6cb4bcd5723 (accessed Sep. 02, 2022).

[2] "Baseball Savant: Trending MLB Players, Statcast and Visualizations," baseballsavant.com. https://baseballsavant.mlb.com/ (accessed Sep. 02, 2022).

[3] Turner, C. (2020, July 20). "The pinch-hitter problem" The Diamond. from https://readthediamond.com/research/the-pinch-hitter-problem

[4] Pathak, S. (2021, July 9). Markov chain algorithm in sports. Medium. from https://medium.com/analytics-vidhya/markov-chain-algorithm-in-sports-a54d086c155e

[5] Statshacker. (2018, August 14). Statistical analysis of the stochastic Markov matrices. statshacker. from http://statshacker.com/blog/2018/06/26/statistical-analysis-of-the-stochastic-markov-matrices/

[6] Beaudoin, David. "Various applications to a more realistic baseball simulator" Journal of Quantitative Analysis in Sports, vol. 9, no. 3, 2013, pp. 271-283

[7] Bukiet, Bruce, et al. "A Markov Chain Approach to Baseball." Operations Research, vol. 45, no. 1, 1997, pp. 14–23. JSTOR.

[8] Belic, Filip Ševerdija, D. Hocenski, Zeljko. (2011). Naive matrix multiplication versus strassen algorithm in multi-thread environment. Tehnicki Vjesnik. 18. 309-314.

[9] Mattson, Tim. (2001). An introduction to openMP. 3-3. 10.1109/CCGRID.2001.923161.