

Computing Equilibria in Bimatrix Games by Parallel Support Enumeration*

Jonathan Widger and Daniel Grosu[†]
Dept. of Computer Science
Wayne State University
5143 Cass Avenue, Detroit, MI 48202 USA.
{jwidger,dgrosu}@cs.wayne.edu

Abstract

We consider the problem of computing all Nash equilibria in bimatrix games (i.e., nonzero-sum two-player non-cooperative games). Computing all Nash equilibria for large bimatrix games using single-processor computers is not feasible due to the exponential time required by the existing algorithms. We consider the use of parallel computing which allows us to solve larger games. We design and implement a parallel algorithm for computing all Nash Equilibria in bimatrix games. The algorithm computes all Nash equilibria by searching all possible supports of mixed strategies. We perform experiments on a cluster computing system to evaluate the performance of the parallel algorithm.

1. Introduction

Game theory studies mathematical models of interaction between rational decision-makers. It represents the standard tool for modeling and analysis in economics and multi-agent systems. The advent of the Internet expanded its use to new areas such as e-commerce systems, network routing, resource allocation in distributed systems, sponsored search auctions and grid computing. This led to the development of Algorithmic Game Theory, which combines algorithmic thinking with game-theoretic concepts [10].

The central solution concept for noncooperative games is that of Nash equilibrium [8, 9]. Nash equilibrium is the solution of a game from which no player can improve her welfare by deviating. It represents a prediction of the strategic behavior of the players in the game. Computing such equilibria is one of the fundamental problems in Algorithmic Game Theory. Its complexity is considered to be the “most important concrete open question at the boundary of

P today” [12].

In this paper we consider the computation of Nash equilibria in bimatrix games, which are noncooperative nonzero-sum two-player games. The existing methods and algorithms for solving bimatrix games can be divided into two categories: (i) algorithms for finding all Nash equilibria; and (ii) algorithms for finding a sample Nash equilibrium.

The simplest algorithm for finding all Nash equilibria is the support enumeration algorithm. This algorithm is searching all the possible pairs of supports of mixed strategies and checks if they satisfy the Nash equilibrium conditions. It takes exponential time since the total number of pairs that need to be explored is exponential. This method is described in [7, 20]. A Mathematica implementation of the support enumeration algorithm is described in [2]. Gambit [6] which is a software tool for solving games also implements this algorithm. Computing all Nash equilibria can also be based on enumerating the vertices of the best response polytopes of the two players. This method is described in [5] and implemented in Gambit [6]. All these implementations are for single processor computers.

Several algorithms for finding a sample Nash equilibrium have been proposed in the past. The most notable is the Lemke-Howson algorithm [4] which has been the standard algorithm used in the last forty years for finding a sample Nash equilibrium in bimatrix games. The algorithm is a complementary pivoting algorithm that solves the linear complementarity problem corresponding to the bimatrix game. For some classes of games the run time of this algorithm is exponential even in the best case [17]. Mixed-integer programming [16] methods have been used to find a sample Nash equilibrium, and found to do well when compared to Lemke-Howson for some games, and worse for others. Simple search algorithms for finding a sample Nash equilibrium have been proposed in [14]. These algorithms are based on support enumeration in which the search is biased towards small supports. Two comprehensive surveys of algorithms for finding equilibria in two player games

*This research was supported in part by NSF grant DGE-0654014.

[†]Corresponding author.

are [19] and [20]. Several algorithms for finding a sample Nash equilibrium in n-player games have been proposed in [3, 14, 18].

NP-completeness is not a suitable concept to characterize the complexity of finding a Nash equilibrium since every game has a Nash equilibrium. It has been proven that finding Nash equilibria even for two-player games is PPAD-complete, where PPAD stands for Polynomial Parity Arguments on Directed graphs [13]. Also it is not known whether a Nash equilibrium can be found in polynomial time in the worst case [13]. On the other hand, it is possible to create bimatrix games in such a way that in the best case it takes exponential time to find a Nash equilibrium using the Lemke-Howson algorithm [17].

The standard software package for generating games to support the performance analysis of game solving algorithms is GAMUT [11]. Gambit [6] is a collection of software tools created to analyze games. Gambit implements almost all the algorithms described above. However, the implementation of these algorithms is sequential.

Our contributions. To the best of our knowledge there are no parallel implementations of algorithms for computing Nash equilibria available to the research community or reported in the literature. The main contribution of this paper is the design and implementation of a parallel algorithm for computing all Nash equilibria in bimatrix games. We implement our algorithm using MPI. We run tests on a grid computing system to study the performance of our implementation in terms of scalability and execution time.

Organization. The paper is structured as follows. In Section 2 we present the basic concepts used in describing our algorithm. In Section 3 we present the sequential algorithm and an example of bimatrix game. In Section 4 we present the parallel algorithm. In Section 5 we investigate the performance of our implementation. In Section 6 we draw conclusions and present future directions.

2. Bimatrix games and equilibria

A bimatrix game is a finite, two-person, non-zero-sum noncooperative game defined as follows.

Definition 1. A bimatrix game $\Gamma(A, B)$ consists of:

- (i) a set of two-players, player 1 called the row-player and player 2 called the column player;
- (ii) a finite set of actions or pure strategies for each player: $M = (s_1, s_2, \dots, s_m)$, the set of m pure strategies of player 1, and $N = (t_1, t_2, \dots, t_n)$, the set of n pure strategies of player 2;
- (iii) the payoff matrices $A \in \mathbb{R}^{M \times N}$ and $B \in \mathbb{R}^{M \times N}$ corresponding to player 1 and player 2.

A mixed strategy for player 1 is an m -vector $x = (x_1, x_2, \dots, x_m)$, where x_i represents the probability of choosing pure strategy s_i . Similarly, a mixed strategy for player 2 is an n -vector $y = (y_1, y_2, \dots, y_n)$, where y_j represents the probability of choosing pure strategy t_j . Thus a mixed strategy for a player is a probability distribution on the set of player's actions. The support of mixed strategy x , denoted by M_x , where $M_x \subset M$, is the set of all actions s_i of player 1 that have positive probability, i.e., $x_i > 0$. Similarly, the support of mixed strategy y , denoted by N_y , where $N_y \subset N$, is the set of all actions t_j of player 2 that have positive probability, i.e., $y_j > 0$.

If x and y are the mixed strategies of player 1 and 2 respectively, then their expected payoffs are given by $x^T A y$ and $x^T B y$, respectively. A best response of player 1 to the mixed strategy y of player 2 is a mixed strategy x that maximizes player 1's payoff, i.e., $x^T A y$. Similarly, the mixed strategy y of player 2 is a best response to the mixed strategy x of player 1 if it maximizes her payoff $x^T B y$.

The solution concept for normal form noncooperative games is that of Nash Equilibrium. The Nash equilibrium in mixed strategies of a bimatrix game is a pair of mixed strategies (x, y) that are best responses to each other. Formally, a Nash equilibrium for two-player games can be defined as follows:

Theorem 1. The mixed strategy (x, y) is a Nash equilibrium of $\Gamma(A, B)$ if and only if

$$\text{for all } s_i \in M_x, (A y)_i = u = \max_{q \in M} \{(A y)_q\} \quad (1)$$

and

$$\text{for all } t_j \in N_y, (x^T B)_j = v = \max_{r \in N} \{(x^T B)_r\} \quad (2)$$

The first condition of this theorem states that a mixed strategy x of player 1 is a best response to mixed strategy y of player 2 if and only if all pure strategies s_i in the support of x are best responses to mixed strategy y . The second condition represents the best response condition corresponding to player 2.

Nash [9] proved that any game with a finite set of players and a finite set of strategies has a Nash equilibrium in mixed strategies. Nash equilibria of a finite bimatrix game are not necessarily unique.

The algorithms that will be presented in this paper compute the Nash equilibria only for non-degenerate games. A non-degenerate bimatrix game is a game in which no mixed strategy of support size k has more than k pure best responses. For such games the Nash equilibrium in mixed strategies is given by strategies that have equal size supports [20].

3. Sequential algorithm for computing equilibria

Nash equilibria of non-degenerate bimatrix games can be computed by enumerating all possible pairs of supports (M_x, N_y) of mixed strategies and checking if a Nash equilibrium exists. Since for a non-degenerate game the mixed strategies of the two players that are candidates for Nash equilibria must have the same support size, the support enumeration algorithm limits the search to only such pairs. The algorithm generates all possible pairs of supports having the same size k , $k = 1, \dots, \min\{m, n\}$. For each such pair of supports it determines the candidate mixed strategy (x, y) . A candidate mixed strategy pair is determined by solving the equations:

$$\sum_{i \in M_x} x_i b_{ij} = v, \quad \text{for } j \in N_y \quad (3)$$

$$\sum_{i \in M_x} x_i = 1 \quad (4)$$

and

$$\sum_{j \in N_y} y_j a_{ij} = u \quad \text{for } i \in M_x \quad (5)$$

$$\sum_{j \in N_y} y_j = 1 \quad (6)$$

The first set of equations determines the mixed strategy x with support M_x of player 1 that makes player 2 indifferent among playing the pure strategies in N_y . That means player 2 obtains the same payoff, v , by playing the pure strategies in N_y if player 1 plays x . The second set of equations is similar but determines the mixed strategy y of player 2.

The above linear equations may have non-unique solutions. This happens for degenerate games because of linear dependencies present in the game's payoff matrices. The algorithm does not provide solutions in the case of degenerate games. If the linear equations do not have a solution then there is no Nash equilibrium for that support pair. The systems of equations can be solved using LU decomposition and back-substitution [15].

If the above equations have (x, y) as the unique solution, then the algorithm checks if the components of x and y are non-negative. They need to satisfy this since they are vectors of probabilities. Finally the algorithm checks if the two mixed strategies satisfy the best response conditions i.e., all pure strategies in the supports M_x and N_y must yield maximum and equal expected payoff for the corresponding player. The mixed strategy pair that satisfies this final condition is a mixed strategy Nash equilibrium for the game. The sequential support enumeration algorithm [20] is given in the following.

Algorithm 3.1. (SUPPORT ENUMERATION)

Input: A bimatrix game: $\Gamma(A, B)$;

Output: All Nash equilibria: (x, y) ;

1. **for** $k = 1, \dots, \min\{m, n\}$ **do**
2. **for each** (M_x, N_y) , $M_x \subseteq M$, $N_y \subseteq N$,
3. $|M_x| = |N_y| = k$ **do**
4. Solve:
5. $\sum_{i \in M_x} x_i b_{ij} = v$ for $j \in N_y$, and
6. $\sum_{i \in M_x} x_i = 1$
7. $\sum_{j \in N_y} y_j a_{ij} = u$ for $i \in M_x$, and
8. $\sum_{j \in N_y} y_j = 1$
9. **if** ($x \geq 0$ **and** $y \geq 0$ **and** conditions (1) and (2)
10. **hold for** x **and** y) **then**
11. output Nash equilibrium (x, y) .

The complexity of the sequential support enumeration algorithm is $O((n+m)^3 \binom{m+n}{n})$. This results from the fact that for a game with $m > n$ there are $\binom{m+n}{n} - 1$ possible pairs (M_x, N_y) that need to be searched. The system of linear equations can be solved in $O((n+m)^3)$ using standard algorithms such as Gaussian elimination or LU decomposition [15]. In the case of square bimatrix games ($m = n$) there are $\binom{2n}{n} - 1$ possible pairs (M_x, N_y) . By using Stirling's formula this is approximately $\frac{4^n}{\sqrt{\pi n}}$. Thus the complexity of the support enumeration algorithm for square games is $O(4^n n^3)$.

Example. As an example we show how the support enumeration algorithm works for the following 3x2 bimatrix game.

$$A = \begin{bmatrix} 4 & 4 \\ 3 & 6 \\ 0 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & 3 \\ 3 & 7 \\ 4 & 2 \end{bmatrix}.$$

Since $m = 3$ and $n = 2$, we need to explore mixed strategies of support size $k = 1$ and $k = 2$. We first consider the supports of size 1, which give us the pure strategy Nash equilibria. The game has only one pure strategy Nash equilibrium given by $((1, 0, 0), (1, 0))$.

Next, we examine the mixed strategies with support size 2. We have three possible pairs of mixed strategies $((x_1, x_2, 0), (y_1, y_2))$, $((x_1, 0, x_3), (y_1, y_2))$ and $((0, x_2, x_3), (y_1, y_2))$.

For the first pair of mixed strategies we have to solve: $4x_1 + 3x_2 = 3x_1 + 7x_2$; $x_1 + x_2 = 1$ and $4y_1 + 4y_2 = 3y_1 + 6y_2$; $y_1 + y_2 = 1$. The solution to these equations is $x_1 = 4/5$, $x_2 = 1/5$ and $y_1 = 2/3$, $y_2 = 1/3$. The vector of expected payoffs to player 1 is $Ay = (4, 4, 7/3)$. The best response condition (1) is satisfied and thus the mixed strategy $((4/5, 1/5, 0), (2/3, 1/3))$ is a Nash equilibrium.

For the second pair of mixed strategies we have to solve: $4x_1 + 4x_3 = 3x_1 + 2x_3$; $x_1 + x_3 = 1$ and $4y_1 + 4y_2 = 7y_2$; $y_1 + y_2 = 1$. The solution to these equations is $x_1 = 4/3$, $x_3 = -1/3$ and $y_1 = 3/7$, $y_2 = 4/7$. The vector x is not a

vector of probabilities and thus there is no Nash equilibrium for this pair of supports.

For the third pair of mixed strategies we have to solve: $3x_2 + 4x_3 = 7x_2 + 2x_3$; $x_2 + x_3 = 1$ and $3y_1 + 6y_2 = 7y_2$; $y_1 + y_2 = 1$. The solution to these equations is $x_2 = 1/3$, $x_3 = 2/3$ and $y_1 = 1/4$, $y_2 = 3/4$. The vector of expected payoffs to player 1 is $Ay = (4, 21/4, 21/4)$. The best response condition (1) is satisfied and thus the mixed strategy $((0, 1/3, 2/3), (1/4, 3/4))$ is a Nash equilibrium.

4. Parallel algorithm

The support enumeration method has great potential for parallelism since the tasks of checking the Nash equilibrium conditions for different support pairs are independent. Once a pair of strategies is generated the computation can proceed independently from the ones corresponding to the other pairs of strategies.

We consider a message-passing system consisting of P processors. To avoid communication overheads we replicate the two game matrices A and B on each processor in the initialization phase of the algorithm. Each processor p generates all the possible pairs of supports, but checks the conditions for Nash equilibrium for only a fraction of them. This way each processor is assigned almost the same number of supports to search and thus we achieve a load balanced execution.

The parallel support enumeration algorithm is given below.

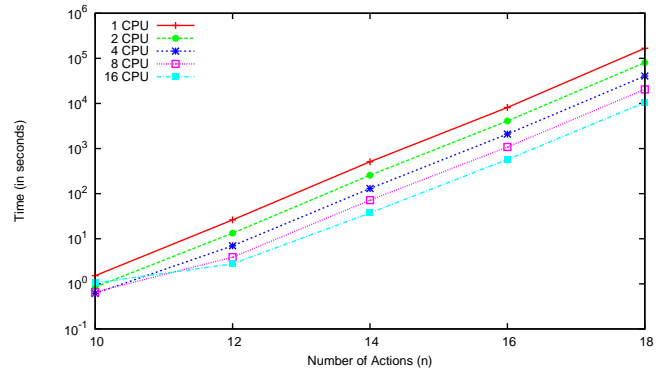
Algorithm 4.1. (PARALLEL SUPPORT ENUMERATION)

Input: A bimatrix game: $\Gamma(A, B)$;
Output: All Nash equilibria: (x, y) ;

1. **for** $p = 0, \dots, P - 1$ **do in parallel**
2. Processor p ;
3. $counter = 0$
4. **for** $k = 1, \dots, \min\{m, n\}$ **do**
5. **for each** $(M_x, N_y), M_x \subseteq M, N_y \subseteq N,$
6. $|M_x| = |N_y| = k$ **do**
7. **if** $(counter \bmod P = p)$ **then**
8. **Solve:**
9. $\sum_{i \in M_x} x_i b_{ij} = v$ for $j \in N_y$, and
10. $\sum_{i \in M_x} x_i = 1$
11. $\sum_{j \in N_y} y_j a_{ij} = u$ for $i \in M_x$, and
12. $\sum_{j \in N_y} y_j = 1$
13. **if** $(x \geq 0$ **and** $y \geq 0$ **and** conditions (1) and (2)
14. **hold for** x and $y)$ **then**
15. **output** Nash equilibrium (x, y) .
16. $counter = counter + 1$

Since each processor executes its task independently of the other processors we expect the algorithm to achieve a speedup close to the ideal speedup of P .

Figure 1. Average execution time vs. number of actions



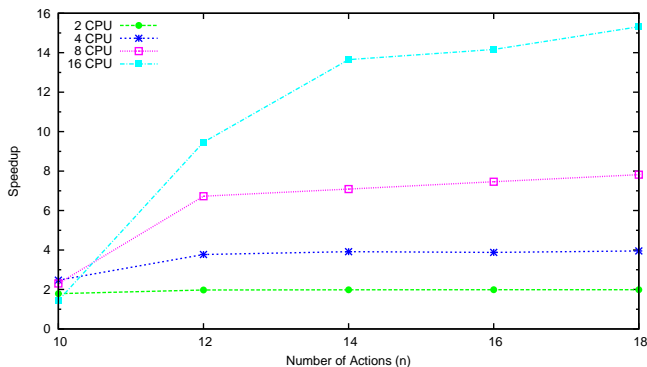
5. Experimental results

We used a cluster within the Wayne State University grid system to conduct the performance analysis of both sequential and parallel algorithms. For our experiments we used sixteen compute nodes with two 2.66GHz Pentium 4 Xeon processors per node. Each of these nodes has 2.5GB of RAM and has access to network storage devices. The nodes are interconnected using a 2Gbit Myrinet network as well as 1Gbit Ethernet. As a programming platform we used MPICH [1] a portable implementation of the Message-Passing Interface standard.

The test games were randomly generated using GAMUT [11]. For our tests we choose the Minimum Effort Game, a classical type of game where the payoff for an action is dependent on the effort associated with the action minus the minimum effort of any player. In this type of game the payoff to one player is given by $a + bM - cE$ where M is the minimum effort of any player in the game, E is the effort of the player, and a, b, c with $b > c$ are constants. The players in the game have the same number of actions. It was chosen as a test game due to its ability to scale in the number of actions. The arguments used to generate the games are: `-int_payoffs -output TwoPlayerOutput -players 2 -actions n -g MinimumEffortGame -random_params`. The value of n determines the number of players' actions and thus the size of the test game.

We selected games of five sizes, given by the number of players' actions: 10, 12, 14, 16, and 18 actions, where each player has the same number of actions. Five configurations of parallel machines were created: 1, 2, 4, 8, and 16 processors. For each of the five game sizes, we randomly generated three games that were used as test games in our experiments. To measure the time taken by the sequential and parallel algorithms we used the command line `time`.

Figure 2. Average speedup vs. number of actions

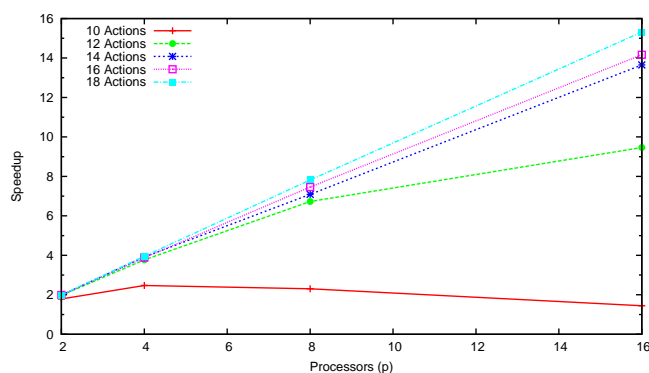


We first investigate the execution time of the sequential and parallel algorithms for different game sizes. Figure 1 gives the average execution time (using a logarithmic scale) versus the number of actions per game. It can be observed that the execution time for a given number of processors grows exponentially with the number of actions and is determined by the total number of support pairs that are searched. As an example for a 10-action game the total number of possible support pairs that are searched by the algorithms is 184,756. When the number of actions increases to 12 the total number of possible support pairs searched by the algorithm increases to 2,704,156. We do not consider games with less than 10 actions because they are small and the amount of necessary computation is dominated by the overheads of parallelization. The speedup that can be achieved for small games is not significant.

The execution time of the sequential algorithm for a 10-action game is about 1.5 seconds, while the execution time for the parallel algorithm with 16 processors is about 1.0 second, a very small improvement. For the 18-action game, the sequential algorithm takes about two days to complete, whereas the parallel algorithm using 16 processors takes about three hours, which represents a significant improvement in the execution time. The data shows that for small games when more processors are used, the overhead of parallelization impacts the overall execution time. For example in the 10-action game the communication overhead is significant enough that the execution time increases when we scale the system to eight and sixteen processors.

In Figure 2 we compare the speedup obtained by the parallel algorithm for games of different sizes. The speedup is defined as the ratio of the serial execution time and the parallel execution time, $S = \frac{T_{sequential}}{T_{parallel}}$. The ideal speedup that can be obtained when parallelizing an algorithm is equal to the number of processors used to execute the par-

Figure 3. Average speedup vs. number of processors

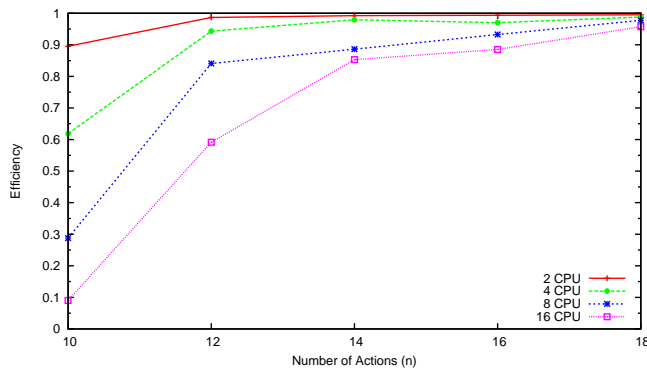


allel algorithm. Compared to the sequential algorithm, the parallel algorithm does well as long as the games are big enough. For the largest game considered in these experiments, the average speedup for the execution on two processors is 1.9899, while the average speedup for execution on sixteen processors is 15.3. This is close to the ideal speedup and is due to the highly parallel nature of the problem. For larger games and a fixed number of processors, it is expected that the speedup obtained by the parallel algorithm asymptotically approaches the number of processors used. The degree at which the speedup is affected is contingent upon the ratio of time spent setting up the communications between nodes and the amount of time required for processing the mixed strategy support pairs at each processor. The ratio is larger for more processors due to the increase in the amount of time spent on initializing the MPI implementation and the decrease in the total amount of work performed by the processors. The problem size has to increase to accommodate a larger set of processors in order to keep the same speedup.

Figure 3, shows the speedup obtained by the parallel algorithm versus the number of processors utilized. The games with the largest number of actions, 14, 16, and 18 lead to a linear dependence between the speedup and the number of processors. Games with fewer actions start to trail off in efficiency when more processors are used. For example in the case of the 10-action game the speedup for 4 processors is 2.47 which is the maximum speedup obtained for this game size. Increasing the number of processors for solving the 10-action game leads to a decreased speedup and thus to performance degradation.

Figure 4 represents the efficiency of the parallel algorithm for games of increasing size. Efficiency is the ratio of the speedup and the number of processors used, $E = \frac{S}{p}$, characterizing how efficiently the parallel implementation

Figure 4. Average efficiency vs. number of actions



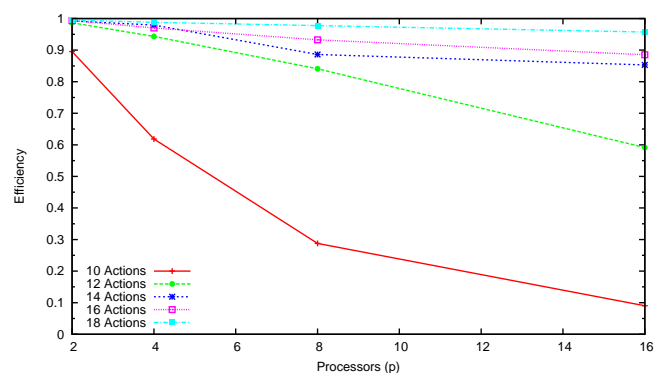
is using the processors. The ideal efficiency is equal to 1. The efficiency of our parallel algorithm increases dramatically for games having more than 12 actions, approaching the ideal efficiency for games with 18 actions. For the 18-action game we considered, the efficiency for all parallel system sizes is greater than 0.95. This tells us that there is very little overhead compared to the amount of computation performed at each processor by our parallel algorithm when the number of actions in the game is more than 12.

Figure 5 shows the efficiency versus the number of processors used. For games with a high number of actions, the efficiency stays fairly constant and near 1 for any number of processors used by the parallel algorithm. A linear decrease in efficiency can be seen for the 12-action game, and a huge decrease for the 10-action game. Again, this is due to the significant amount of overhead induced by the initialization and starting up the parallel implementation. We expect that the efficiency for larger games to be close to 1 for a fixed number of processors.

6. Conclusion

We designed and implemented a parallel algorithm for computing all Nash equilibria in bimatrix games by support enumeration. We analyzed its performance for different game sizes and different number of processors. The analysis shows that the algorithm scales very well with the number of processors used in the implementation. Solving games having hundreds or more actions in a reasonable amount of time would require using hundreds of processors. We are planning to perform experiments using more than one hundred processors. For future work we plan to provide parallel versions of other types of methods for finding Nash equilibria. For this new parallel versions we plan to develop and investigate load allocation schemes which will

Figure 5. Average efficiency vs. number of processors



lead to efficient execution on large scale parallel systems.

References

- [1] MPICH - A Portable Implementation of MPI. <http://www-unix.mcs.anl.gov/mpi/mpich1/>.
- [2] J. Dickhaut and T. Kaplan. A program for finding nash equilibria. *Mathematica Journal*, 1(4):87–93, 1991.
- [3] S. Govindan and R. Wilson. A global newton method to compute nash equilibria. *Journal of Economic Theory*, 110:65–86, 2003.
- [4] C. E. Lemke and J. T. Howson. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12(4):413–423, 1964.
- [5] O. L. Mangasarian. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12(4):778–780, 1964.
- [6] R. D. McKelvey, McLennan, and T. Turocy. Gambit: Software tools for game theory, Version 0.2007.01.30. Available at <http://gambit.sourceforge.net>, 2007.
- [7] R. B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, Cambridge, MA, 1991.
- [8] J. F. Nash. Equilibrium points in n -person games. *Proc. Nat'l Academy of Sciences of the United States of Am.*, 36(1):48–49, Jan. 1950.
- [9] J. F. Nash. Non-cooperative games. *Annals of Math.*, 54(2):286–295, 1951.
- [10] N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani. *Algorithmic Game Theory*. Cambridge University Press, New York, 2007.
- [11] E. Nudelman, J. Wortman, Y. Shoham, and K. Leyton-Brown. Run the gamut: A comprehensive approach to evaluating game-theoretic algorithms. In *Proc. of 3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 880–887, August 2004.
- [12] C. Papadimitriou. Algorithms, games and the internet. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 749–753, July 2001.
- [13] C. H. Papadimitriou. *The Complexity of Finding Nash Equilibria*, chapter 2, *Algorithmic Game Theory*, (eds. N. Nisan,

- T. Roughgarden, E. Tardos, and V. Vazirani), pages 29–52. Cambridge Univ. Press, 2007.
- [14] R. Porter, E. Nudelman, and Y. Shoham. Simple search methods for finding a nash equilibrium. In *Proc. of the 19th National Conference on Artificial Intelligence*, pages 664–669, July 2004.
- [15] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, New York, 2007.
- [16] T. Sandholm, A. Gilpin, and V. Conitzer. Mixed-integer programming methods for finding nash equilibria. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 495–501, July 2005.
- [17] R. Savani and B. von Stengel. Hard-to-solve bimatrix games. *Econometrica*, 74(2):397–429, 2006.
- [18] G. van der Laan, A. J. J. Talman, and L. van der Heyden. Simplicial variable dimension algorithms for solving the nonlinear complementarity problem on a product of unit simplices using a general labelling. *Mathematics of Operations Research*, 12(3):377–397, 1987.
- [19] B. von Stengel. *Computing equilibria for two-person games*, chapter Handbook of Game Theory, Vol. 3 (eds. R. J. Aumann and S. Hart), pages 1723–1759. North-Holland, 2002.
- [20] B. von Stengel. *Equilibrium computation for two-player games in strategic and extensive form*, chapter 3, Algorithmic Game Theory, (eds. N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani), pages 53–78. Cambridge Univ. Press, 2007.