

Computing Equilibria in Bimatrix Games by Parallel Vertex Enumeration

Jonathan Widger
Department of Computer Science
Wayne State University
5143 Cass Avenue
Detroit MI 48202, USA
Email: dw0587@wayne.edu

Daniel Grosu
Department of Computer Science
Wayne State University
5143 Cass Avenue
Detroit MI 48202, USA
Email: dgrosu@cs.wayne.edu

Abstract—Equilibria computation is of great importance to many areas such as economics, control theory, and recently computer science. We focus on the computation of Nash equilibria in two-player general-sum normal form games, also called bimatrix games. One efficient method to compute these equilibria is based on enumerating the vertices of the best response polyhedrons of the two players and checking the equilibrium conditions for every pair of vertices. We design and implement a parallel algorithm for computing Nash equilibria in bimatrix games based on vertex enumeration. We analyze the performance of the proposed algorithm by performing extensive experiments on a grid computing system.

Keywords—game theory; Nash equilibrium; parallel algorithm; bimatrix game

I. INTRODUCTION

Game theory, the study of strategic interactions among rational agents, has had a great impact on many sciences such as economics, evolutionary biology, control theory, and recently, computer science. The most notable applications of game theory to computer science are in networking, electronic commerce, multiagent systems, sponsored search auctions, and resource allocation in grid computing systems [1].

The central solution concept in noncooperative game theory is the Nash equilibrium. Nash equilibrium can be viewed as a situation in which no player can do better by unilaterally changing her strategy. Nash [2], [3] established the existence of equilibria for any finite game. Computing such equilibria is very important for practitioners since in many cases equilibria represent predictions of the strategic behavior of the players. Computing the Nash equilibria of small games can be performed feasibly with current computing technology using standard sequential algorithms. Real-life scenarios, however, are modeled using large scale games that cannot be feasibly solved using sequential algorithms. This is because the current algorithms require exponential time to compute all Nash equilibria of a game. Recently, it has been proved that finding Nash equilibria for two-player games is PPAD-complete, where PPAD stands for Polynomial Parity Arguments on Directed graphs [4]. It is not known whether a Nash equilibrium can be found in

polynomial time in the worst case [4].

In this paper, we focus on the problem of computing all Nash equilibria in bimatrix games. The simplest algorithm for computing all Nash equilibria is the support enumeration algorithm [5]. The idea behind this algorithm is to search all the possible pairs of supports of mixed strategies and to check if they satisfy the Nash equilibrium conditions. This requires exponential time, since the total number of pairs that need to be explored is exponential in the number of players' strategies. This method is common knowledge, but it was first described in [5]. A Mathematica implementation of the support enumeration algorithm is described in [6]. Gambit [7], which is a software tool for solving games, implements this algorithm. All these implementations are for single processor computers. In our previous work [8], we designed and implemented a parallel support enumeration algorithm for computing all Nash equilibria.

Another method for computing all Nash equilibria is based on enumerating the vertices of the best response polyhedrons of the two players. This method is described in [9], [10]. Avis [11] designed the lexicographical reverse search library, *lrslib*, for vertex enumeration of polyhedra, which has recently been used in a sequential algorithm for finding Nash equilibria in bimatrix games. In a more recent paper, Avis *et al.* [12] presented two sequential algorithms for computing Nash equilibria in bimatrix games based on *lrslib* and on the method of enumerating extreme equilibria. ZRAM [13] was developed by Marzetta as a portable library for parallel search using an old implementation of *lrs* library. However, it does not appear to have been maintained in the last several years to reflect changes and optimizations in *lrslib*. In this paper, we provide a scalable parallel vertex enumeration algorithm for computing all Nash equilibria in bimatrix games.

Finding a sample Nash equilibrium in bimatrix games is another problem that was investigated extensively in the past [14], [15]. The Lemke-Howson algorithm [16] is the most notable algorithm for computing a sample equilibrium. It 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]. A comprehensive survey of algorithms for finding equilibria in two player games is [10].

A. Our contributions

We propose a parallel vertex enumeration algorithm for computing all Nash equilibria in bimatrix games. The vertex enumeration at each processor is based on the lexicographical search algorithm. The parallel algorithm is implemented using MPICH [18] and Irslib [11]. We run several experiments on a grid system and investigate the performance of our algorithm.

B. Organization

The paper is structured as follows. In Section II, we present the basic concepts of bimatrix games and equilibria. In Section III, we describe the sequential vertex enumeration algorithm for computing Nash equilibria in bimatrix games. In Section IV, we present the design of our parallel vertex enumeration algorithm. In Section V, we investigate the performance of our parallel algorithm. In Section VI, we draw conclusions and present some ideas for future work.

II. BIMATRIX GAMES AND EQUILIBRIA

In this section we introduce the basic concepts of bimatrix game theory. In presenting the concepts we use notation similar to that used in [10]. 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 [2]. 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 of a bimatrix game can be characterized as follows:

Theorem 1. [10] 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 [3] 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.

Example 1. We illustrate the concepts defined above considering the following 3×2 bimatrix game.

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

Player 1 has three pure strategies represented by the three rows, while player 2 has two strategies represented by the two columns. If player one is playing the first strategy and the second player is playing her second strategy then the payoff to player 1 is 4, and the payoff to player 2 is 3. Examples of mixed strategies are $x = (4/5, 1/5, 0)$ for player 1, and $y = (2/3, 1/3)$ for player 2. The payoffs to (x, y) are $x^T A y = 4$ for player 1, and $x^T B y = 19/5$ for player 2. The game has one pure Nash equilibrium given by the pair of strategies $((1, 0, 0), (1, 0))$, and two mixed strategy equilibria given by $((4/5, 1/5, 0), (2/3, 1/3))$ and $((0, 1/3, 2/3), (1/4, 3/4))$.

The best response conditions in Theorem 1 together with the equations defining the mixed strategies as probabilities define the so called best response polyhedrons. The best response polyhedrons for the two players are defined as follows:

$$P_1 = \{(x, v) \in \mathbb{R}^M \times \mathbb{R} \mid x \geq \mathbf{0}, \mathbf{1}^T x = 1, B^T x \leq \mathbf{1}v\}$$

$$P_2 = \{(y, u) \in \mathbb{R}^N \times \mathbb{R} \mid A y \leq \mathbf{1}u, y \geq \mathbf{0}, \mathbf{1}^T y = 1\}$$

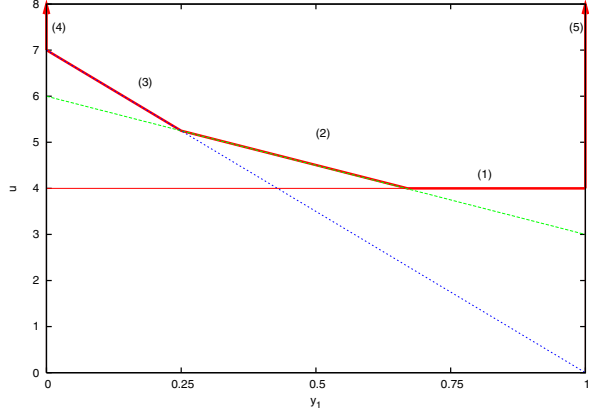


Figure 1. Player 2's best response polyhedron

In the case of player 1's polyhedron, P_1 , the first two inequalities say that x is a vector of probabilities, while the last inequality says that v is as large as the expected payoff to player 2 for each pure strategy of player 2. Similarly, for P_2 , the first inequality says that u is as large as the expected payoff to player 1 for each pure strategy of player 1, while the last two inequalities say that y is a vector of probabilities.

Example 2. Player 2's best response polyhedron for the game in Example 1 is given by:

$$P_2 : \begin{cases} 4y_1 + 4y_2 \leq u & (1) \\ 3y_1 + 6y_2 \leq u & (2) \\ 0y_1 + 7y_2 \leq u & (3) \\ y_1 \geq 0 & (4) \\ y_2 \geq 0 & (5) \\ y_1 + y_2 = 1 \end{cases}$$

This polyhedron is shown in Figure 1. Similarly, we can define the best response polyhedron for player 1, but because of the space restrictions we do not present it here.

In order to describe the Nash equilibrium conditions and the vertex enumeration algorithm we need to define labels for the vertices of these polyhedrons. A point (y, u) of P_2 has label k if the k -th inequality defining P_2 is binding. The inequalities defining P_2 are ordered such that the inequalities corresponding to player 1's strategies s_1, s_2, \dots, s_m appear first, followed by the inequalities corresponding to positive probabilities for y_j . In Example 2, the labels are indicated at the right of each inequality defining P_2 . Similarly, a point (x, v) of P_1 has label k if the k -th inequality defining P_1 is binding. The inequalities defining P_1 are ordered such that the inequalities corresponding to positive probabilities for x_i appear first, followed by the inequalities corresponding to player 2's strategies t_1, t_2, \dots, t_n . In the example above, $(y_1, y_2, u) = (2/3, 1/3, 4)$ has labels $\{1, 2\}$. This labeling means that pure strategies s_1 and s_2 are best responses to mixed strategy $y = (2/3, 1/3)$. The expected payoff to player 1 corresponding to this vertex is 4.

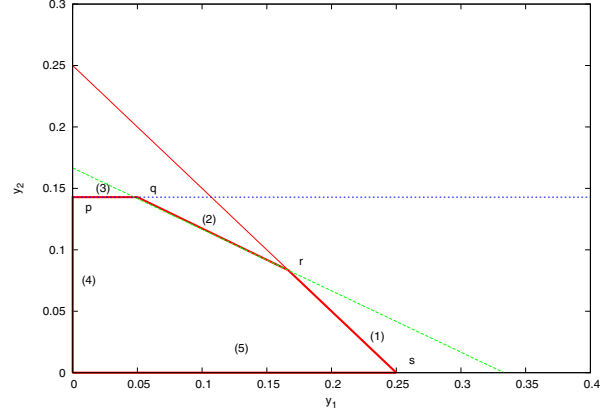


Figure 2. Player 2's best response polytope

A completely labeled pair $((x, v), (y, u)) \in P_1 \times P_2$ is a pair in which every label appears as a label of either (x, v) or (y, u) . A completely labeled pair corresponds to a situation in which each pure strategy is a best response, or has probability zero. This is equivalent to the best response condition in Theorem 1, i.e., x and y are best responses to each other. Thus, a completely labeled pair $((x, v), (y, u)) \in P_1 \times P_2$ corresponds to Nash Equilibrium (x, y) . An algorithm that computes all Nash equilibria can be based on this fact. It explores all the possible vertex pairs and checks if they are completely labeled.

In order to simplify the constraints defining the polyhedrons for the two players we will eliminate the payoff variables u and v . This is possible if payoffs u and v are positive. We thus, transform the polyhedrons into polytopes (bounded polyhedrons) by dividing each inequality $x^T B \leq 1v$ in P_1 by v and replace x_i/v by x_i , and by dividing each inequality $Ay \leq 1u$ in P_2 by u and replace y_j/u by y_j . We obtain the following best response polytopes for the two players:

$$Q_1 = \{x \in R^M | x \geq 0, B^T x \leq 1\}$$

$$Q_2 = \{y \in R^N | Ay \leq 1, y \geq 0\}$$

Example 3. Player 2's best response polytope for the bimatrix game in Example 1 is given by:

$$Q_2 : \begin{cases} 4y_1 + 4y_2 \leq 1 & (1) \\ 3y_1 + 6y_2 \leq 1 & (2) \\ 0y_1 + 7y_2 \leq 1 & (3) \\ y_1 \geq 0 & (4) \\ y_2 \geq 0 & (5) \end{cases}$$

Player 1's best response polytope is given by:

$$Q_1 : \begin{cases} x_1 \geq 0 & (1) \\ x_2 \geq 0 & (2) \\ x_3 \geq 0 & (3) \\ 4x_1 + 3x_2 + 4x_3 \leq 1 & (4) \\ 3x_1 + 7x_2 + 2x_3 \leq 1 & (5) \end{cases}$$

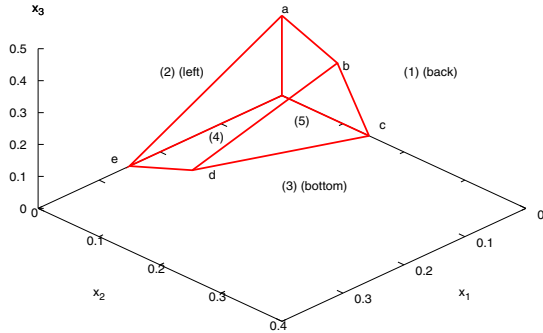


Figure 3. Player 1's best response polytope

The polytopes for the two players are shown in Figure 2 and 3. In the following section we describe a sequential algorithm that is searching for completely labeled pairs in the players' polytopes and finds the Nash equilibria.

III. SEQUENTIAL VERTEX ENUMERATION ALGORITHM FOR COMPUTING EQUILIBRIA

The sequential vertex enumeration algorithm, given in Algorithm 1, finds the Nash equilibria of bimatrix games by enumerating all the possible pairs of vertices of the two best response polytopes and checking if the pairs are completely labeled [10]. Once a completely labeled pair is found it outputs the corresponding mixed strategy Nash equilibrium. Since the vertices are not probability vectors, we divide them by $\mathbf{1}^T x$ and respectively $\mathbf{1}^T y$ to obtain the mixed strategies. The algorithm also works with polyhedra and not only with polytopes.

The time complexity of the sequential vertex enumeration algorithm is $O(2.6^n)$ for square games (i.e., games in which both players have the same number n of possible actions). This algorithm is faster than the support enumeration algorithm which requires $O(4^n)$ time for square games [10].

Example 4. In this example, we show how the algorithm finds all the Nash equilibria of the game given in Example 1 whose corresponding polytopes are given in Figure 2 and 3. The vertices of Q_2 are: $(0, 0)$ with labels $\{4,5\}$; p with labels $\{3,4\}$; q with labels $\{2,3\}$; r with labels $\{1,2\}$; and

Algorithm 1 Sequential Vertex Enumeration

```

1: for all vertices  $x \in Q_1 - \{0\}$  do
2:   for all vertices  $y \in Q_2 - \{0\}$  do
3:     if  $(x, y)$  completely labeled then
4:       output Nash equilibrium  $(x/(\mathbf{1}^T x), y/(\mathbf{1}^T y))$ 
5:     end if
6:   end for
7: end for

```

s with labels $\{1,5\}$. The vertices of Q_1 are: $(0, 0, 0)$ with labels $\{1,2,3\}$; a with labels $\{1,2,4\}$; b with labels $\{1,4,5\}$; c with labels $\{1,3,5\}$; d with labels $\{3,4,5\}$; and, e and with labels $\{2,3,4\}$. The algorithm enumerates these vertices and finds three pairs of completely labeled vertices: (i) (e, s) which corresponds to pure Nash equilibrium $((1, 0, 0)(1, 0))$; (ii) (d, r) which corresponds to mixed Nash equilibrium $((4/5, 1/5, 0)(2/3, 1/3))$; and (iii) (b, q) which corresponds to mixed Nash equilibrium $((0, 1/3, 2/3)(1/4, 3/4))$.

The lexicographical reverse search algorithm [11] can be used to enumerate the vertices of each polyhedron. The reverse search operates on a system of linear inequalities defining a d -dimensional polyhedron. A vertex in this polyhedron is determined by the indices of d bounding inequalities. By using a linear objective function and the simplex method, a path can be found between adjacent vertices. When the objective function is maximized, the path terminates and the next vertex has been found. This method determines a spanning tree of polyhedron's vertices and explores it using depth-first search to iterate across all of the vertices. At each node in the spanning tree during the enumeration process, a *dictionary* is created which corresponds to the currently enumerated vertex. The dictionaries are defined in detail in [11]. To simplify our description we consider that a dictionary is associated with a vertex. A dictionary can be used to stop and restart the search process at any time as well as be modified to alter the parameters of the search. We use these techniques in the design of our parallel vertex enumeration algorithm presented in the next section.

IV. PARALLEL VERTEX ENUMERATION ALGORITHM FOR COMPUTING EQUILIBRIA

The parallel vertex enumeration algorithm is based on a master-worker model. At the beginning of the computation, one of the workers, called seeder, has a special role that is to 'seed' the workers. The seeder starts enumerating the spanning tree of polyhedron P_1 's vertices up to a given depth d . The seeder sends the subtrees corresponding to vertices at depth d to the workers which will perform the enumeration of vertices in these subtrees. The workers have two roles: producers and consumers. When a worker has the producer role it will enumerate the vertices in the subtrees received from the seeder and send the vertices found to the master. The master collects the vertices of P_1 from the workers. These vertices are distributed in equal chunks back to the workers that will have this time the role of consumers. A worker with the role of consumer gets a set of vertices of P_1 from the master and then performs vertex enumeration on P_2 's vertices checking for complete labeling. When a complete labeled pair is found, the corresponding Nash equilibrium is sent to the output. We decided to use a seeder in the design of our parallel algorithm in order to increase the parallelism and the worker throughput in data preparation.

Algorithm 2 Master

```
1:  $type_0 \leftarrow 0$ ;  $type_1 \leftarrow 1$ ;  $type_{i=2,\dots,P} \leftarrow 2$ 
2:  $queue \leftarrow \emptyset$ 
3:  $done \leftarrow \text{false}$ 
4: while not done do
5:    $receive(node_i, msg, tag)$ 
6:   if tag is vertex_set then
7:      $insert(queue, msg)$ 
8:   end if
9:   if (growth(queue)  $\geq g$  or length(queue)  $\geq q$ ) or
   (length(queue)  $> 0$  and  $type_1 \neq 1$ ) then
10:     $type_i \leftarrow 3$ 
11:     $w \leftarrow remove(queue, w_s)$ 
12:     $send(node_i, w, work\_unit)$ 
13:  else if type_1 = 1 then
14:     $type_i \leftarrow 2$ 
15:     $send(node_i, \emptyset, producer)$ 
16:  else
17:     $type_i \leftarrow 0$ 
18:     $send(node_i, \emptyset, done)$ 
19:  end if
20:  if  $\sum_{i=1}^n type_i = 0$  then
21:     $done \leftarrow \text{true}$ 
22:  end if
23: end while
```

We now describe each of the algorithm components in detail. Upon initialization each node in the parallel machine is assigned a unique identifier (id) from 0 to P , where P is the total number of machines used. The identifier determines the role of the node; master has $id = 0$, seeder has $id = 1$, and workers have ids form 2 up to P .

The master, whose id is 0, is tasked with three jobs, to collect and distribute sets of vertices of polyhedron P_1 which we call work units, to keep track of the status of each node, and to determine whether a worker should be producing or consuming work units. The master maintains an array, $type$, containing the type of the nodes. The type of a node with id i ($type_i$) can be: 0 = none, 1 = seeder, 2 = producer, and 3 = consumer. The master also maintain a queue ($queue$) containing vertices from player 1's polyhedron, P_1 .

The following notation is used in the description of the master component of the algorithm: w_s - number of vertices that are sent to a worker as a work unit; g - threshold on the growth of the $queue$ before assigning workers to process work units; q - threshold on the number of vertices in the queue before assigning workers to process work units; msg - container for incoming messages; tag - type of message received; and w - a set of P_1 's vertices. We also use the following functions to describe the master algorithm: $insert(queue, set)$ - inserts a set of vertices into $queue$; $remove(queue, num)$ - removes num items from $queue$; $growth(queue)$ - finds the rate at which $queue$ is growing; $length(queue)$ - returns the current number of items in $queue$.

The master algorithm is given in Algorithm 2. The master loops through received messages, and if the message is a

Algorithm 3 Seeder

```
1:  $V \leftarrow \emptyset$ 
2:  $D_1 \leftarrow lrs(P_1)$ 
3:  $d \leftarrow \text{maxdepth}(D_1)$ 
4:  $setdepth(D_1, d * r)$ 
5: while  $D_1 \leftarrow \text{nextdictionary}(D_1)$  do
6:   if  $\text{currentdepth}(D_1) = \text{maxdepth}(D_1)$  then
7:      $D_1^i \leftarrow D_1$ 
8:      $setdepth(D_1^i, d)$ 
9:      $receive(worker_i, msg, tag)$ 
10:     $send(worker_i, D_1^i, seed)$ 
11:   else
12:      $v_1 \leftarrow \text{getvertex}(D_1)$ 
13:      $V \leftarrow V \cup \{v_1\}$ 
14:   end if
15: end while
16:  $send(master, V, \text{vertex\_set})$ 
```

set of vertices (i.e., $tag = \text{vertex_set}$) it adds them to the work queue (lines 6-8). Then, the master determines the role of the worker's next iteration depending on the growth of the queue, the size of the queue, and if there is any more expected work to be produced (i.e., checks if the seeder is not active, which is equivalent to $type_1 \neq 1$). If there is more work expected to be produced by the seeder, that means the seeder is still active (i.e., $type_1 = 1$), the master determines the role of the worker as producer (lines 13-15). Finally, the algorithm finishes each pass of the main loop checking whether execution has finished by taking the sum of the elements of the array containing the types of the nodes. When all the types are set to 0, the vertex enumeration is complete.

The seeder is a specialized worker created at initialization, whose id is 1. The role of the seeder is to start the vertex enumeration on player 1's polyhedron, P_1 , but terminate it at a given depth of the spanning tree as determined by the ratio r of the maximum depth d (i.e., at depth $r \times d$). Due to the nature of the lexicographical reverse search algorithm, the spanning tree corresponding to P_1 can be split into many smaller spanning trees P_1^i , each of which can be processed in parallel to search for vertices. This is done by setting a new depth at which to search for vertices in P_1^i and using the results of that search as a basis of a new search at the original maximum depth to find vertices. At tree nodes between the root and the cutoff depth, vertices defined within the dictionary are saved until the seeding process is complete. Upon completion of distributing subtrees to the workers, the seeder becomes a worker to clear the queue of vertices held by the master.

The following notation is used in the description of the seeder algorithm: r - the ratio of the depth at which the seeder ends its search process; D_1 - dictionary for player 1 as defined by $lrslib$; d - the original depth of the spanning tree created from P_1 ; D_1^i - a derived dictionary from P_1 ; v_1 - a vertex from P_1 ; and V - a set to contain vertices

Algorithm 4 Worker

```
1:  $D_2 \leftarrow \text{lrs}(P_2)$ 
2:  $result \leftarrow \emptyset$ 
3:  $\text{send}(\text{master}, \emptyset, \emptyset)$ 
4:  $done \leftarrow \text{false}$ 
5: while not done do
6:    $\text{receive}(\text{master}, \text{msg}, \text{tag})$ 
7:   if tag is work_unit then
8:     for all  $v_1$  in msg do
9:        $D'_2 \leftarrow D_2$ 
10:       $\text{remove}(D'_2, v_1)$ 
11:      while  $D'_2 \leftarrow \text{nextdictionary}(D'_2)$  do
12:         $v_2 \leftarrow \text{getvertex}(D'_2)$ 
13:        if complete( $v_1, v_2$ ) then
14:           $result \leftarrow result \cup \{(v_1, v_2)\}$ 
15:          print ( $v_1, v_2$ )
16:        end if
17:      end while
18:    end for
19:     $\text{send}(\text{master}, result, result)$ 
20:  else if tag is producer then
21:     $\text{send}(\text{seeder}, \emptyset, \text{request\_for\_work})$ 
22:     $\text{receive}(\text{seeder}, D_1^i, \text{tag})$ 
23:    while  $D_1^i \leftarrow \text{nextdictionary}(D_1^i)$  do
24:       $v_1 \leftarrow \text{getvertex}(D_1^i)$ 
25:       $result \leftarrow result \cup \{v_1\}$ 
26:    end while
27:     $\text{send}(\text{master}, result, \text{vertex\_set})$ 
28:  else if tag is done then
29:     $done \leftarrow \text{true}$ 
30:  end if
31:   $result \leftarrow \emptyset$ 
32: end while
```

enumerated by the seeder.

The following functions are used in the description of the seeder: $\text{nextdictionary}(\text{dictionary})$ - returns the next dictionary, or false if there is none; $\text{maxdepth}(\text{dictionary})$ - gets the maximum search depth of the spanning tree defined by the dictionary; $\text{setdepth}(\text{dictionary}, d)$ - sets the maximum search depth of the spanning tree defined by the dictionary; $\text{currentdepth}(\text{dictionary})$ - gets the current depth of the dictionary in the spanning tree; and $\text{getvertex}(\text{dictionary})$ - returns the vertex corresponding to the dictionary.

The algorithm for the seeder is given in Algorithm 3. Lines 1 through 4 define environment variables for the ratio of maximum depth to search, initialize player 1's data (P_1) in lrslib , save the maximum depth of the original data, and modify the maximum depth to only perform a partial search of the spanning tree corresponding to D_1 . In lines 5 through 14, the seeder explores the vertices of P_1 's spanning tree up to the cutoff depth defined by $r \times d$. If the current vertex is at the cutoff depth then the corresponding dictionary, D_1^i , is sent to a worker that requested work (lines 9-10). If the current vertex is not at the cutoff depth then the vertex is placed into V . The set of vertices enumerated by the seeder, V is then sent to the master for further processing. After the seeder ends 'seeding' the computation, it becomes a regular

worker and executes the algorithm for the worker.

Workers have two jobs, either taking a subtree specified by D_1^i from the seeder and enumerate its vertices, or taking a number of P_1 's vertices from the master and checking for complete labeling against P_2 's vertices. At initialization, all the workers are tasked with processing subtrees from the seeder. When the vertices are found, the labels of each vertex are sent in bulk to the master.

The following notation is used in the description of the worker algorithm: D_2 - dictionary for player 2 as defined by lrslib ; $result$ - set to contain work units (P_1 's vertices) or completely labeled pairs of vertices; v_2 - vertex from player 2; and D'_2 - a modified dictionary from player 2. The following functions are used in the description of the worker: $\text{remove}(\text{dictionary}, \text{vertex})$ - removes labels found in vertex from dictionary ; and $\text{complete}(v_1, v_2)$ - determines whether a vertex pair is completely labeled.

The algorithm for the worker is given in Algorithm 4. Line 1 sets up lrs with player 2's data in D_2 , followed by asking the master for work. The algorithm enters a loop and depending on one of two types of messages received on line 6 determines the next action. If it is a work unit ($\text{tag} = \text{work_unit}$), the worker iterates through each vertex contained within it. For each vertex v_1 , it makes a copy of D_2 to D'_2 and removes the labels found in the vertex to reduce the search space in D'_2 . Then, for each vertex found in D'_2 , if the vertex contains the remainder of the labels, the equilibrium condition is satisfied and the vertex pair is added to the results. Finally, on line 19, the worker sends the results to the master. If the message received in line 6 tells the worker to become a producer ($\text{tag} = \text{producer}$), the worker requests from the seeder the next subtree to be searched (specified by D_1^i). In a similar fashion to the seeder, it finds the vertices of D_1 and puts them into a result set to send to the master, in line 27. The final message received is from the master ($\text{tag} = \text{done}$), telling the worker to exit. After each iteration the result set, having already been sent to the master, is emptied and if the worker has not been told to exit, it listens for the next incoming message.

V. EXPERIMENTAL RESULTS

We ran our experiments on sixteen computers (nodes) which are part of the Wayne State University Grid. Each node contains two Intel Xeon processors at 2.6GHz, has 2.5GB of RAM, and a 1GB/second network interconnect running Linux [19]. The parallel programming environment used to implement the parallel algorithm is MPICH a portable implementation of MPI [18]. The test games were generated using GAMUT [20], a defacto standard utility for generating games. The type of game we chose for our tests is the Minimum Effort Game, having between 14 and 30 actions per player, and randomly generated coefficients. 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

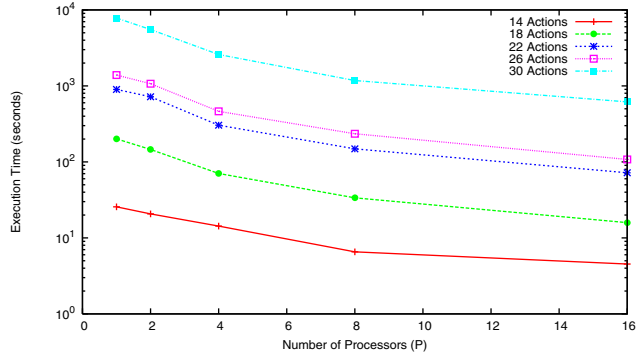


Figure 4. Execution time vs. number of processors

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. The value of n determines the number of players' actions, and thus, the size of the test game. In our experiments we choose the depth ratio $r = 0.5$ (that means the seeder searches half the depth of player 1's spanning tree), the master's queue size of 500, the growth limit of 20 vertices per second, and the work unit size of 10 vertices.

In Figure 4, we show the execution time for different game sizes as a function of the number of processors used. As the number of nodes in the parallel machine increases, we obtain a significant decrease in the total execution time. In the case of games with 30 actions per player, we observe a steady decrease in execution time, from 7816.543 seconds or 2.15 hours, using the sequential algorithm, to 616.256 seconds or about 10 minutes, using 16 nodes. In cases where the total computational requirement is not significant, such as for a game having 14 actions per player, there is a slower decrease in execution time as the network communication dominates the computation. The proposed parallel vertex enumeration algorithm is much faster than our previous parallel support enumeration algorithm [8]. As an example, for games with 18 actions, the parallel support enumeration algorithm running on 16 nodes required about two days to complete, while the parallel vertex enumeration required less than 10 minutes to complete.

In Figure 5, we show the speedup obtained by our parallel algorithm. The speedup is defined as the ratio of the execution time of the sequential algorithm to the execution time of the parallel algorithm. We obtained a linear speedup for games with more than 18 actions per player. For games with fewer than 18 actions per players the speedup decreases significantly due to the overhead induced by the communication. For example in the case of games with 14 actions per player the maximum speedup of 5.8 is reached when running on 16 processors. Increasing the number of processors for games with 14 actions or less leads to a very small increase in speedup, and thus, to lower efficiency.

In Figure 6, we show the efficiency of our implementation,

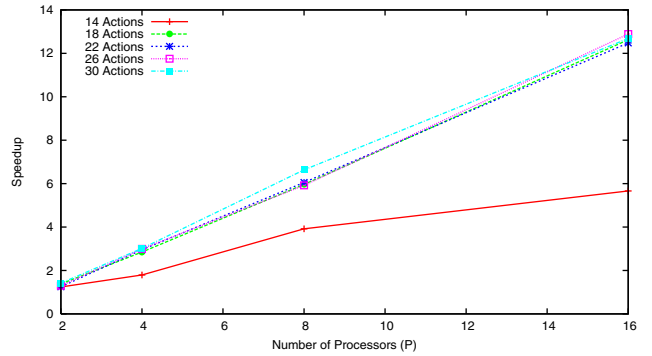


Figure 5. Speedup vs. number of processors

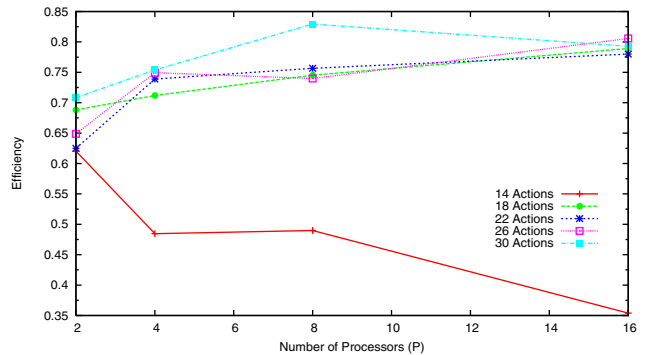


Figure 6. Efficiency vs. number of processors

where efficiency is defined as the ratio of speedup to the number of processors used. For games with more than 18 actions per players and systems with more than 4 processors the efficiency is between 0.7 and 0.85. For these games the efficiency increases when the number of nodes in the system increases. This can be attributed to the greater degree of tuning available between producers and consumers of work units in the algorithm, as with using fewer nodes the efficiency is not as great even for larger games. For a system with 16 processors, we expect to obtain an efficiency close to 0.8 in solving games with more than 30 actions per player.

VI. CONCLUSIONS AND FUTURE WORK

We designed and implemented a parallel vertex enumeration algorithm that computes all Nash equilibria of large bimatrix games in reasonable amount of time using parallel systems with few number of processors. By using a balancing mechanism we were able to efficiently achieve good speedup despite the sequential nature of a depth-first search algorithm. The performance results show that the algorithm is scalable achieving good efficiency for large games.

Future research goals include creating a portable package of parallel algorithms for finding Nash equilibria. We also plan to design heuristics for finding the optimal cutoff depth of the spanning tree explored by the seeder in order

to further optimize the distribution of work, especially on heterogeneous machines.

ACKNOWLEDGMENT

This research was supported in part by NSF grant DGE-0654014. We thank Prof. David Avis for his very helpful suggestions.

REFERENCES

- [1] N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani, *Algorithmic Game Theory*. New York: Cambridge University Press, 2007.
- [2] J. F. Nash, "Equilibrium points in n -person games," *Proc. Nat'l Academy of Sciences of the United States of Am.*, vol. 36, no. 1, pp. 48–49, Jan. 1950.
- [3] —, "Non-cooperative games," *Annals of Math.*, vol. 54, no. 2, pp. 286–295, 1951.
- [4] C. H. Papadimitriou, *The Complexity of Finding Nash Equilibria*. Cambridge Univ. Press, 2007, ch. 2, Algorithmic Game Theory, (eds. N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani), pp. 29–52.
- [5] R. B. Myerson, *Game Theory: Analysis of Conflict*. Cambridge, MA: Harvard University Press, 1991.
- [6] J. Dickhaut and T. Kaplan, "A Program for Finding Nash Equilibria," *Mathematica Journal*, vol. 1, no. 4, pp. 87–93, 1991.
- [7] 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.
- [8] J. Widger and D. Grosu, "Computing Equilibria in Bimatrix Games by Parallel Support Enumeration," in *Proc. of the 7th International Symposium on Parallel and Distributed Computing*. IEEE Computer Society Press, July 2008, pp. 250–256.
- [9] O. L. Mangasarian, "Equilibrium Points of Bimatrix Games," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 4, pp. 778–780, 1964.
- [10] B. von Stengel, *Equilibrium computation for two-player games in strategic and extensive form*. Cambridge Univ. Press, 2007, ch. 3, Algorithmic Game Theory, (eds. N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani), pp. 53–78.
- [11] D. Avis, "*lrs*: A revised implementation of the reverse search vertex enumeration algorithm," Available at <http://cgm.cs.mcgill.ca/avis/doc/avis/Av98a.ps>, 1999.
- [12] D. Avis, G. D. Rosenberg, R. Savani, and B. von Stengel, "Enumeration of Nash equilibria for two-player games," *Economic Theory*, vol. 42, 2010.
- [13] A. Marzetta, "Zram: A library of parallel search algorithms and its use in enumeration and combinatorial optimization," Ph.D. dissertation, ETH Zurich, 1998.
- [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*, July 2004, pp. 664–669.
- [15] 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*, July 2005, pp. 495–501.
- [16] C. E. Lemke and J. T. Howson, "Equilibrium Points of Bimatrix Games," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 4, pp. 413–423, 1964.
- [17] R. Savani and B. von Stengel, "Hard-to-Solve Bimatrix Games," *Econometrica*, vol. 74, no. 2, pp. 397–429, 2006.
- [18] "MPICH - A Portable Implementation of MPI," <http://www-unix.mcs.anl.gov/mpi/mpich1/>.
- [19] "Wayne State University Grid," <https://www.grid.wayne.edu/>.
- [20] 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*, August 2004, pp. 880–887.