Parallel Nested Monte-Carlo Search

Tristan Cazenave Nicolas Jouandeau

***Abstract*—We address the parallelization of a Monte-Carlo search algorithm. On a cluster of 64 cores we obtain a speedup of 56 for the parallelization of Morpion Solitaire. An algorithm that behaves better than a naive one on heterogeneous clusters is also detailed.**

***Index Terms*—Monte-Carlo, Search, Parallelization, Morpion Solitaire**

I. Introduction

Monte-Carlo methods can be used to search problems that have a large state space and no good heuristics. Nested Monte-Carlo search [7] improves Monte-Carlo search using a lower level Monte-Carlo Search to choose move at the upper level. For problems that do not have good heuristics to guide the search, the use of nested levels of Monte-Carlo search amplifies the results of the search and makes it better than a simple Monte-Carlo search. A similar algorithm has already been applied with success to Morpion Solitaire [6]. We address the parallelization of the Nested Monte-Carlo Search algorithm on a cluster.

Section 2 describes related work, section 3 explains the sequential Nested Monte-Carlo Search algorithm, section 4 presents two parallel algorithms: the Round-Robin algorithm and the Last-Minute algorithm, section 5 details experimental results.

II. Related works

Parallel algorithms have been developed for many different metaheuristics [1], [19], [2].

Rollout algorithms were first proposed by Tesauro and Galperin for improving a Backgammon program [20]. The idea of a rollout is to improve a heuristic playing games that follow the heuristic and using the result of these games to evaluate moves. Rollouts were applied to different optimization problems [4], [3], [12], [18].

Nested rollouts were found effective for the game of Klondike solitaire [21]. The base level uses a domain specific heuristic to guide the samples. Nested rollouts have also been used for Thoughtful Solitaire, a version of Klondike Solitaire in which the locations of all cards is known. In this case they were used with heuristics that change with the stage of the game [5].

Recently there has been interest in the parallelization of Monte-Carlo tree search, especially for the game of Go [8], [14], [9], [10], [17].

Morpion Solitaire is an NP-hard puzzle [11]. A move consists in adding a circle such that a line containing five circles can be drawn. Lines can either be horizontal, vertical or diagonal. The starting position already contains circles disposed as in figure 1. In the disjoint version a circle cannot be a part of two lines that have the same direction. The best human score at Morpion Solitaire disjoint version is 68 moves [11]. The previous best computer score was 79 obtained with Simulated Annealing [16]. A reflexive Monte-Carlo algorithm was shown to be effective for Morpion Solitaire [6]. Reflexive Monte-Carlo search is close in spirit to nested rollouts except that the base level plays random games and does not follow a heuristic.

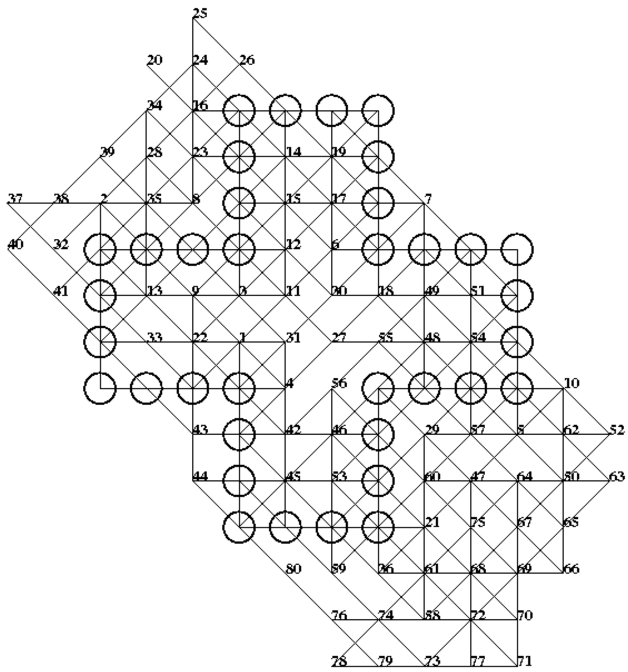

Fig. 1. A world record found by Parallel Nested Monte-Carlo Search at Morpion Solitaire disjoint version

Guerriero and Mancini have proposed two parallel strategies for rollout algorithms [15]. They were tested on the Traveling Salesman Problem (TSP) and the Sequential Ordering Problem (SOP). With their speculative strategy they obtained a modest average speedup of 2.53 on 8 processors for SOP and TSP. They obtained speedup ranging from 3.89 to 6.64 on 8 processors depending on the size of the neighborhood for SOP.

In this paper we propose the parallelization of a nested rollout algorithm on a cluster.

III. The sequential algorithm

The basic sample function just plays a random game from a given position:

LAMSADE, Université Paris-Dauphine
Place Maréchal de Lattre de Tassigny, 75775 Paris Cedex 16, France
email: cazenave@lamsade.dauphine.fr
Université Paris 8, LIASD
2 rue de la liberté, 93526, Saint-Denis, France
email: n@ai.univ-paris8.fr

```
int sample (position)
while not end of game
  play random move
return score
```

The score is the score of the game in the terminal position. For example at Morpion Solitaire the goal is to play as many moves as possible, so the score is the number of moves played in the game. In other games where the algorithm is of interest, the score can be computed completely differently. The idea of the score function is that the algorithm tries to find the sequence of moves that maximizes it.

The nested rollout function plays a game, choosing at each step of the game the move that has the highest score of the lower level nested rollout. A level 1 rollout uses the sample function to choose its moves. The $argmax_m$ command sends back the move that returns the best score of a lower level search, over all possible moves:

```
int nested (position, level)
best score = -1
while not end of game
  if level is 1
    move = argmax_m (sample (
                  play (position, m)))
  else
    move = argmax_m (nested (
      play (position, m), level - 1))
  if score of move > best score
    best score = score of move
    best sequence = seq. after move
  bestMove = move of best sequence
  position = play (position,bestMove)
return score
```

IV. Parallel algorithms

In order to parallelize nested rollouts we define four types of processes: the root process, the median node processes, the dispatcher process and the client processes. These processes work at different levels of nesting: the root process at the first level (the highest level of nesting), the median process at the second level and the client processes at the third level. The root process plays a game at the first level and calls the median processes to play games at the second level. The median processes ask the client processes to play games at the third level in parallel. The dispatcher process is used to tell median nodes which clients to use.

A. The Round-Robin algorithm

Figure 2 details the Round-Robin algorithm. There are four possible communications. The first one (fig. 2(a)) consists in a message from the root node to a median node that asks the median node to perform a nested search at the lower level.

During the second communication (fig. 2(b)), the median node asks to the dispatcher which clients it should use, once it has received a client, it asks this client to perform a nested search at the lower level.

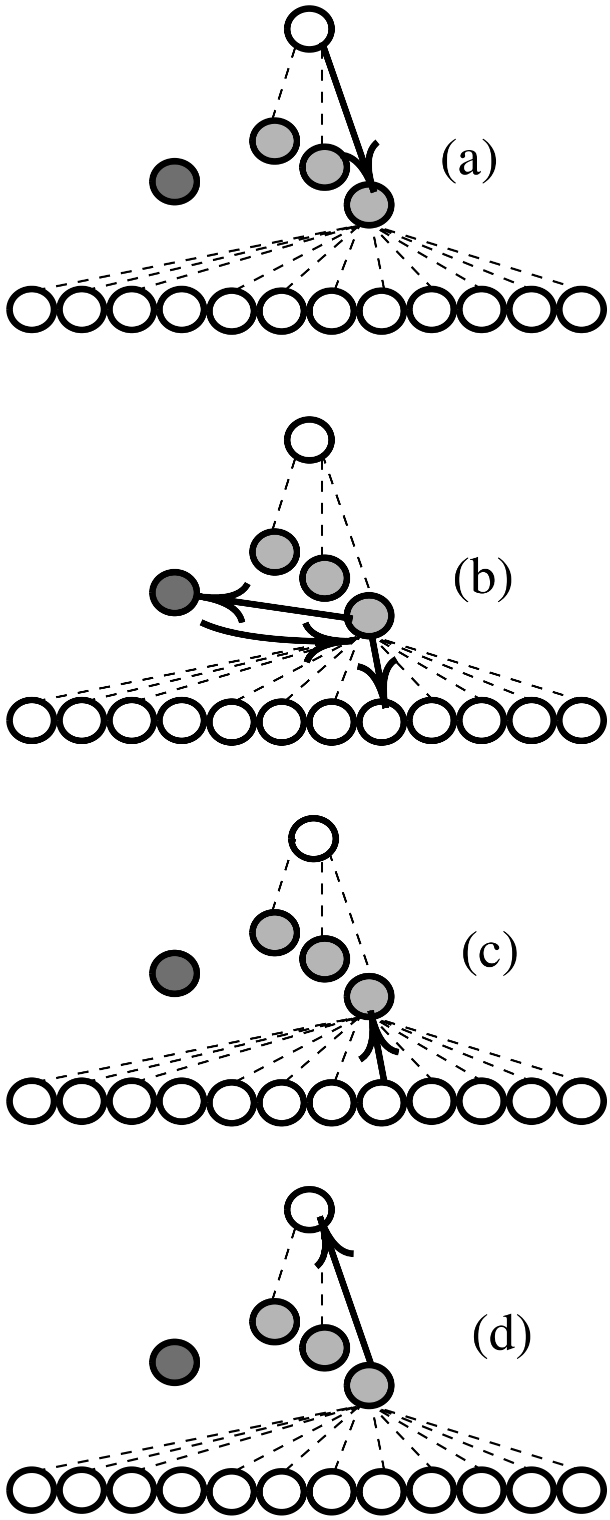

Fig. 2. Communication between processes during the Round-Robin algorithm

The third communication (fig. 2(c)) occurs when a client of a median node has finished its search. It then sends back the result to the median node that has asked for this search. Once the median node has all its results, it can choose the best move, play it and continue its game going back to the second communication. When the game is over, it uses the fourth communication (fig. 2(d)) which consists in sending the result of the game to the root node.

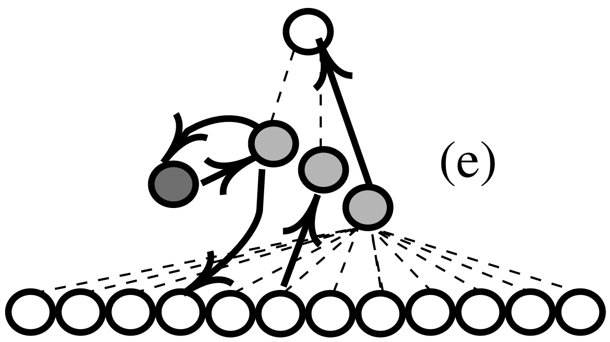

Fig. 3. Parallel communications that can occur during the Round-Robin algorithm

Figure 3 shows that the second, third and fourth communications can occur in parallel.

The root process plays a game until the end at the highest level. The number of median nodes is greater than the number of possible moves. Median nodes are mainly used to dispatch the computation at a lower level on the client nodes, they are not used for long computation. Starting from the current position of the game, the root process tries each possible move and sends the resulting position to a different median node. Then it waits for all answers from median nodes previously chosen. Once it has all answers, it can choose the move that has the highest score, play it and loop until the end of the game.

The pseudo code for the root process is :

```
while not end of game
  node = first median node
  for m in all possible moves
    p = play (position, m)
    send p to node
    node = next median node
  for m in all possible moves
    receive score from node
  position = play (position,
                   move with best score)
return score
```

The Round-Robin dispatcher consists in choosing the next client in the list of clients for each request. It receives messages from median nodes that ask it on which client to send their computation. It simply sends back clients one after another, always in the same order.

The code for the Round-Robin dispatcher is:

```
client = first client
while true
  receive median node from any median node
  send client to median node
  if client is last client
    client = first client
  else
    client = next client
```

A median process receives a position to play from the root process. At each step of a game, it tries all possible moves and asks the dispatcher which client to use to evaluate each move. It then waits for the client node from the dispatcher, and once it has received it, it sends to the client node the position to evaluate. After having sent all the positions (one for each possible move), it waits for all the corresponding answers from the clients. Each answer consists in a score. When it has received all the scores, it chooses the move that has the highest score and plays it. Then it loops until the end of the game. Eventually, it sends back the score of the game to the root process.

The code for a median process is:

```
while true
  receive position from root process
  while not end of game
    for m in all possible moves
      p = play (position, m)
      send self id and
        number of moves played in p
        to dispatcher
      receive client from dispatcher
      send p to client
    for m in all possible moves
      receive score from client
    position = play (position,
                     move with best score)
  send score to root
```

A client process waits for a position from a median node. When it receives the position, it plays a nested rollout at a predefined level, and sends back the resulting score to the median node. In the case of the Last-Minute algorithm it also warns the dispatcher that it is available, sending its own identifier.

The code for a client process is:

```
while true
  receive position from median node
  score = nestedRollout (position, level)
  if LastMinute
    send self node to dispatcher
  send score to median node
```

B. The Last-Minute algorithm

Figure 4 shows the difference between the Round-Robin algorithm and the Last-Minute algorithm. Instead of only sending back the result of a search to its median node (fig. 2(c)), a client also warns the dispatcher that it is free (fig. 4(c')). The dispatcher maintains a list of free clients and a list of jobs. Jobs are ordered by expected computation time. The expected computation time is estimated with the number of moves already played in a game. When a request is received by the dispatcher, either there are free clients and it sends back the first free client, or there are no free clients and the job is added to the list of jobs. When it is notified by a client that the client is free, either it sends this client to the median node of the longest expected job,

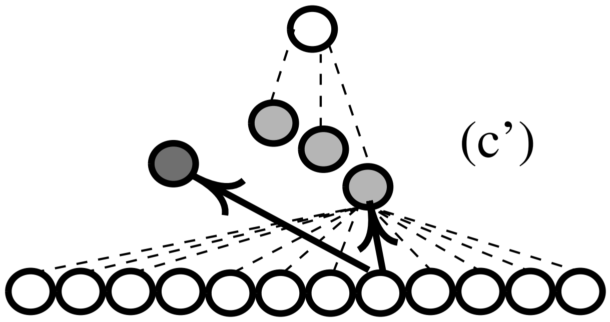

Fig. 4. Communication between processes during the Last-Minute algorithm

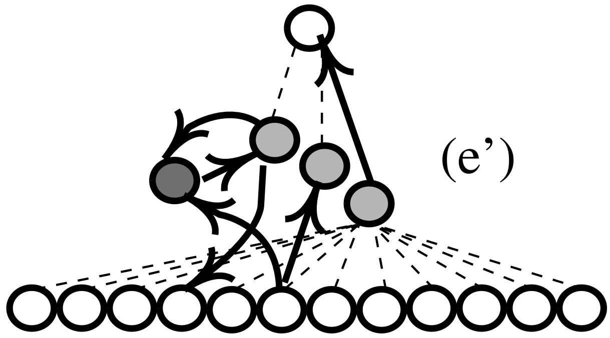

Fig. 5. Parallel communications that can occur during the Last-Minute algorithm

or if there are no available jobs it adds it to the list of free clients. The other communications are similar to the Round-Robin algorithm.

Figure 5(e') shows that communications can occur in parallel.

The Last-Minute dispatcher waits for a node from any client. If it receives a client node, it means that this client node is waiting for a new job, therefore if there are pending jobs it sends the job with the smallest number of moves to this client. If there are no pending job, it simply adds the client to the list of free clients. If it receives a median node, it also receives the number of moves of the position to analyze. If there are no free clients, it adds the job to the list of pending jobs. If there is a free client it sends the free client to the median node and removes the client from the list of free clients.

The pseudo-code for the Last-Minute dispatcher:

```
listFreeClients = all Clients
jobs = empty list
while true
  receive node from any node
  if node is a client node
    add node to listFreeClients
    if jobs is not empty
      find j in jobs with
        the smallest number of moves
      send j.sender to the node
      remove j from jobs
      remove node from listFreeClients
  else if node is a median node
    receive number of moves from node
    addNewJob = true
    if listFreeClients is empty
      add {node, number of moves} to jobs
    else
      client = first element of listFreeClients
      send client to node
      remove client from listFreeClients
```

V. Experimental Results

Our cluster is composed of 20 1.86 GHz dual core PCs, 12 2.33 GHz dual core PCs and one quad core server connected with a Gigabit network. We used message passing with Open MPI [13] as parallel programming model. Open MPI is designed to achieve high performance computing on heterogeneous clusters. All communications are done with the global communicator MPI_COMM_WORLD. Each node runs two client processes. Processes are created at the beginning of the execution, via the use of the master-slave model. The server runs the root process as well as all the median processes and the dispatcher. Most of the computation is performed by the clients. We run the 40 median processes on the server and the client nodes on the dual core PCs.

All experiments use Morpion Solitaire disjoint model. All the time results are a mean over multiple runs of each algorithm, except for results in parenthesis which were run only once. The standard deviation is given between parenthesis after the time results. The algorithms were tested on playing only the first move of a game, and on playing an entire game. All experiments consist in testing the algorithms at level 3 and 4 of nesting. Each rollout needs a time that is slightly different from others since random games inside each rollout can have different lengths. Times taken by two rollouts can be different. Standard deviations show these times variations.

The results for the sequential algorithm are given in table I. We can observe that level 4 takes approximately 207 times more time than level 3. One rollout takes approximately 9 times more time than the first move.

level	first move		one rollout	
3	08m03s	(19s)	1h07m33s	(42s)
4	28h00m06s	(58m55s)	(09d18h58m)	

TABLE I

Times for the sequential algorithm

Table II gives the times for playing the first move with the Round-Robin algorithm at level 3 and 4. The speedup of the algorithm for 64 clients is 56 (in fact it should be a little less since the time with one client is on a 1.86 GHz PC), if we use the ratio of the mean cluster frequency on

clients	level 3		level 4	
64	10s	(1s)	33m11s	(1m33s)
32	20s	(2s)	1h04m44s	(3m02s)
16	37s	(5s)	(2h10m)	
8	01m11s	(8s)	—	
4	02m22s	(11s)	—	
1	09m07s	(28s)	(29h56m14s)	

TABLE II
First move times for the Round-Robin algorithm

clients	level 3		level 4	
64	01m32s	(5s)	4h10m09s	(24m04s)
32	02m43s	(16s)	6h58m21s	(52m42s)
16	05m35s	(40s)	—	
8	11m33s	(1m34s)	—	
4	19m51s	(3m34s)	—	
1	1h31m40s		—	

TABLE V
Rollout times for the Last-Minute algorithm

clients	level 3		level 4	
64	01m52s	(8s)	5h09m16s	(5m40s)
32	03m08s	(26s)	(6h31m)	
16	05m22s	(29s)	—	
8	10m18s	(1m21s)	—	
4	21m41s	(3m13s)	—	
1	1h26m28s		—	

TABLE III
Rollout times for the Round-Robin algorithm

clients	alg	level 3		level 4	
16x4+16x2	LM	14s	(2s)	28m37s	(1m30s)
16x4+16x2	RR	16s	(2s)	45m17s	(1m19s)
8x4+8x2	LM	18s	(3s)	58m21s	(2m44s)
8x4+8x2	RR	25s	(2s)	1h24m11s	(3m24s)

TABLE VI
First move times on an heterogeneous cluster

the single client frequency we obtain $r = ((20 \times 1.86 + 12 \times 2.33)/32)/1.86 = 1.09$. So the speedup should rather be closer to $56/1.09 = 51$. The result for 32 clients is obtained using only 1.86 GHz PCs, and this time the speedup is 29.8. Concerning level 4 the speedup is 28.50 for 32 clients.

Table III gives the times for playing a rollout with the Round-Robin algorithm at level 3 and 4. The speedup of the algorithm for 64 clients is 44.

Table IV gives the results of the Last-Minute algorithm for the first move at levels 3 and 4. Results are similar to the Round-Robin algorithm at level 3. The speedup for level 4 is 30 which is a little higher than the Round-Robin algorithm.

Table V gives the results of the Last-Minute algorithm for rollouts at levels 3 and 4. Results are slightly better than the Round-Robin algorithm.

Table VI compares the two algorithms with an heterogeneous repartition (where 16x4+16x2 means that there are 16 PCs with 4 clients and 16 PCs with 2 clients and where 8x4+8x2 means there are 8 PCs with 4 clients and 8 PCs with 2 clients). At level 4 the Last-Minute algorithm (LM) has better results than the Round-Robin algorithm (RR).

clients	level 3		level 4	
64	09s	(2s)	27m20s	(1m22s)
32	19s	(1s)	59m44s	(2m21s)
16	37s	(4s)	(2h05m17s)	
8	01m12s	(5s)	—	
4	02m23s	(4s)	—	
1	09m30s	(21s)	(33h06m57s)	

TABLE IV
First move times for the Last-Minute algorithm

The Last-Minute algorithm gives better results than the Round-Robin algorithm in an heterogeneous environment. The speedup of the Last-Minute algorithm for rollouts of level 4 with 64 clients is approximately 56 which is quite good.

Running the algorithm at level 4 on our cluster, we have discovered two new sequences of 80 moves which is the current world record.

VI. Conclusion

We have presented two algorithms that parallelize Nested Monte-Carlo Search on a cluster. The speedup for 64 clients is approximately 56 for Morpion Solitaire which is a problem with a large state space and no good known heuristic. The Last-Minute algorithm is more adapted to heterogeneous clusters. The parallel algorithm run at level 4 has found sequences of length 80 which is the current world record at Morpion Solitaire disjoint version (see for example figure 1).

References

[1] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms.* Wiley, 2005.

[2] Enrique Alba, El-Ghazali Talbi, and Albert Y. Zomaya. Nature-inspired distributed computing. *Computer Communications*, 30(4):653–655, 2007.

[3] Dimitri P. Bertsekas and David A. Castañon. Rollout algorithms for stochastic scheduling problems. *J. Heuristics*, 5(1):89–108, 1999.

[4] Dimitri P. Bertsekas, John N. Tsitsiklis, and Cynara Wu. Rollout algorithms for combinatorial optimization. *J. Heuristics*, 3(3):245–262, 1997.

[5] R. Bjarnason, P. Tadepalli, and A. Fern. Searching solitaire in real time. *ICGA Journal*, 30(3):131–142, 2007.

[6] T. Cazenave. Reflexive monte-carlo search. In *Computer Games Workshop*, pages 165–173, Amsterdam, The Netherlands, 2007.

[7] T. Cazenave. Nested Monte-Carlo search. In *IJCAI 2009*, Pasadena, USA, July 2009.

[8] T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Computer Games Workshop 2007*, pages 93–101, Amsterdam, The Netherlands, June 2007.

[9] T. Cazenave and N. Jouandeau. A parallel monte-carlo tree search algorithm. In *Computers and Games*, LNCS, pages 72–80, Beijing, China, 2008. Springer.

[10] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008.

[11] E. D. Demaine, M. L. Demaine, A. Langerman, and S. Langerman. Morpion solitaire. *Theory Comput. Syst.*, 39(3):439–453, 2006.

[12] M. Mancini F. Guerriero and R. Musmanno. New rollout algorithms for combinatorial optimization problems. *Optimization Methods and Software*, 17:627–654, 2002.

[13] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[14] Sylvain Gelly, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud, and Yann Kalemkarian. The parallelization of monte-carlo planning. In *ICINCO*, 2008.

[15] F. Guerriero and M. Mancini. Parallelization strategies for rollout algorithms. *Computational Optimization and Applications*, 31(2):221–244, 2005.

[16] H. Hyyro and T. Poranen. New heuristics for morpion solitaire. Technical report, University of Tampere, Finland, 2007.

[17] H. Kato and I. Takeuchi. Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08)*, November 2008.

[18] Nicola Secomandi. Analysis of a rollout approach to sequencing problems with stochastic routing applications. *J. Heuristics*, 9(4):321–352, 2003.

[19] Alexandru-Adrian Tantar, Nouredine Melab, and El-Ghazali Talbi. A comparative study of parallel metaheuristics for protein structure prediction on the computational grid. In *IPDPS*, pages 1–10, 2007.

[20] G. Tesauro and G. Galperin. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems 9*, pages 1068–1074, Cambridge, MA, 1996. MIT Press.

[21] X. Yan, P. Diaconis, P. Rusmevichientong, and B. Van Roy. Solitaire: Man versus machine. In *Advances in Neural Information Processing Systems 17*, pages 1553–1560, Cambridge, MA, 2005. MIT Press.