Replacement assignment for Parallel Programming, summer 2005
D.C. van Moolenbroek <dcvmoole@cs.vu.nl> (#1275151)

Introduction

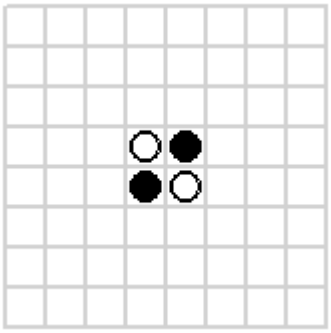
Othello is a simple game with few rules [10], but playing it well is very difficult – there is a good reason that the slogan on the official game box is "A Minute To Learn, A Lifetime To Master". In order to get a bit of a feeling I started by writing a 4x4-board bruteforcer which found the best solution for 4x4 Othello in under a second, then I noted that bruteforcing 6x6 was already infeasible for regular PCs within reasonable time, so clearly bruteforcing 8x8 would be out of the question.

The next best thing is to compute the best next move by evaluating searched boards and using this information to look further in a specific direction. So instead, I've written a program that decides the best move for a given starting board and search depth by evaluating the boards it finds; in summary, I've taken the base from the MTD(f)/Awari implementation that comes with the Ibis distribution, the evaluation function from a third-party Othello program, and started working from there to get a reasonably fast, not too weak, parallelized Othello implementation on top of Ibis/Satin.

The sequential version

Board storage

There are several ways to store an Othello board in memory, and although this may seem a trivial aspect, it is actually quite important.



The first observation is that for an 8x8 board, there are 64 positions where each position can have the state BLACK, WHITE, or EMPTY. This means there are potentially 3^{64} ($\sim 10^{30}$) boards, although not all boards can be reached from the game's starting board shown in the picture. For example, when a position is occupied (i.e. either BLACK or WHITE), it can never become empty again later, so because of the initial position, the boards that involve one or more empty squares in the middle are unreachable. There are many more boards that will never occur in a game, but generalizing those into a set of rules is hard if not impossible.

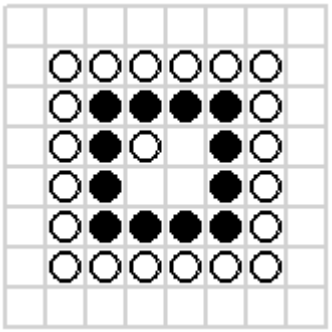
Being able to have fast access to any position on a board is obviously a requirement, but minimizing the size that a board takes up would also be nice, for two reasons: boards are frequently sent over the network, and the transposition table has to store a representation of the board for each entry (to be able to do a perfect equality test – note that MTD(f)/Awari does this in an at best questionable way!) as well. Possibilities include matrix storage (*byte[8][8]*, this is *really* inefficient in Java); array storage (*byte[64]*, much faster) and bit storage (storing each board position's state in two bits – this way a board takes up four 32-bit integers).

For the board class, I have decided to go with array storage, because of its speed, for the transposition table I've used bit storage because of its size. Furthermore, I have decided to have black always move next during the game, so after a move, the stones on the board are inverted and it's black's turn again. This takes away the need to store whose turn it is (along with each board *and* each transposition table entry), and simplifies the functions that look for, and apply, moves.

Searching the game tree

To perform the actual game tree search, the MTD(f) algorithm is currently the most suitable algorithm to perform game searches in several games including Othello [8]. In order not to needlessly duplicate efforts of implementing this algorithm, I have reused the implementation from the MTD(f)/Awari example in the Ibis source tree, including the iterative deepening framework around it. However, the iterative deepening code would reset the MTD(f) pivot value to 0 for each iteration; this was fine for the Awari solver that it came with, but does not work well with the evaluation system that I've used in my Othello version – therefore I've altered the

iterative deepening code to reuse the pivot from the last iteration, an approach that is also suggested in the MTD(f) algorithm description itself [8]. This simple change reduced computation times significantly.



As for each job, children are generated and placed in an array, it was necessary to know how many children can potentially be generated from a node. The theoretical maximum that I could come up with is 31, and can be seen in the picture on the right. Even though this board can not be reached from the normal starting position, I have no way of finding out the actual maximum, so I've chosen to go with a safe array size of 32.

The evaluation function

The ability to properly evaluate a certain board is what makes or breaks a game search program like this one. Obviously lacking the experience and game insight required to construct a proper evaluation function for Othello, I had to resort to taking the evaluation function from an existing Othello implementation – coming up with one myself was simply not feasible. One implementation I found that had a moderately strong and yet reasonably simple evaluation function is Marvin [12], which already was written in Java, and released under the GNU Public License. It uses a combination of a number of evaluation sub-functions, which are well-known ways to evaluate Othello boards [1, 11, 13]:

Stone score: a score based on the number of stones for each player; more stones for you than for the opposite player is better. This obviously holds true for any end position (when the game ends, the player with the most stones is the winner), but it may not be that good of a measurement for mid-game boards: having fewer stones there can actually be an advantage. Also, the number of stones owned by each player may change rapidly during the game, especially close the end.

Square value: a score based on who occupies certain important positions of the board. Corner positions are especially important, as you they can not be retaken by the enemy and generally provide safe access to two edges at once. Similarly, the position on the diagonal one position away from each corner value potentially provides the enemy access to the corner, so this is generally a bad position to occupy. The square value is good for mid-game and end-game evaluation, but is not very meaningful at the start of the game (since noone owns any positions on or close to the edges).

Mobility: the number of moves that you can make, versus the number of moves that your opponent can make. Being able to attack your opponent in more ways is advantageous, having one or zero possible moves means you're forced to do a move or skip a turn, which is generally a bad thing.

Potential mobility: this measures the number of potential moves, that is, the number places where one of your opponent's stones is next to an empty square. This means that at some point, you could fill the empty square and potentially take over the opponent's stone along with it. Having lots of enemy stones 'exposed' this way is a good thing.

There are many more sub-evaluations that an evaluation function can be composed of, many of the more advanced ones involving huge precomputed tables. I've deliberately stayed away from those, in order not to make the game excessively complicated and unintellegible (take a look at the source code of Logistello [3] to see what I mean!). I did attempt to implement an additional feature as suggested in [13]: edge scoring, where a score for each of the four board edges was looked up in a precomputed table. Although obtaining the edge score is not at all an expensive operation, composing an edge table (for each possible edge, i.e. consisting of $3^8 = 6561$ entries) that actually improved the search (I've tried two approaches, a bruteforce approach and the one suggested in [13]) turned out to be too difficult for me, so I've had to abandon this attempt.

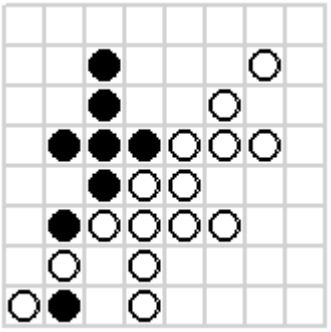
Exactly how strong the current evaluation function is, is hard to tell. Even rather skilled Othello players are said to be easily outskilled by their own programs, and from there it takes alot of expertise to observe the behaviour of the program and actually improve things. There actually is a way to measure the strength of an Othello program, though by comparing it to other Othello programs, using the Generic Game Server's Othello server GGS/os [2], which is the successor to the IOS system that is referred to in a lot of older documents. This is one thing that I would have liked to try, but didn't get around to.

Anyhow, Marvin's scoring system was way too fine-grained for the MTD(f) algorithm: the huge score differences between boards cause the pivot to change a lot. To overcome this, I decided to simply divide the outcome of Marvin's evaluation by eight, obviously resulting in much more coarse-grained values. Although this made pretty much every run at least 50% faster, it does decrease the accuracy of the search a bit; since I don't

know how good Marvin's evaluation function really is anyway, I found this to be an acceptable trade-off. Instead of dividing the final score by eight, I could also have changed the evaluation subfunctions themselves to produce lower values, but not changing the subfunctions themselves has the advantage that they can still be used for move ordering without loss of precision – see the next section.

Move ordering

An important factor for the efficiency of the game search is the choice which of the possible moves on a specific board is tried first: if the first checked move leads to good result, the others won't have to be checked. Therefore, move ordering plays an important role in the search. It is also one that requires balancing: the move ordering function should be sufficiently strong to make a good prediction about which move is best (limiting the number of boards visited), but it iterates over all new child moves that are generated, so it should also be fast enough not to cause a greater performance penalty than can be gained from it. Marvin did not use move ordering at all, so I decided to come up with my own.



The most obvious way to do the move ordering is by using (a combination of) parts of the evaluation function of the generated children. The following tests were conducted by performing a search to depth 13 from the normal Othello starting position (on my own machine though, not on the DAS2 cluster) and to depth 11 from the board in the picture. Since the move ordering is (mostly) unrelated to parallelization, I performed the tests with the sequential version only.

From the initial position:

Move ordering by	Running time (sec)	Boards visited
(None)	1027.806	4,383,996
Stone score	206.645	1,029,845
Square score	290.034	1,284,190
Mobility score	703.624	1,197,055
Potential mobility score	2431.821	7,833,472
Stone score + square score	202.854	853,809

From the mid-game board:

Move ordering by	Running time (sec)	Boards visited
(None)	779.396	3,750,913
Stone score	575.967	3,558,618
Square score	273.581	1,377,694
Mobility score	1858.020	3,649,926
Potential mobility score	261.665	933,746
Stone score + square score	298.481	1,352,399

At first I used only the four basic parts of the evaluation function; after seeing the results, I also tried combining the stone and square scores, the two fastest evaluation subfunctions, and this had good results for both boards - so this is what I've used in the final implementation. However, it is very well possible that more fine-tuned combinations of the evaluation subfunctions results in a more generally applicable move ordering function.

Before the tests above, I did not use full sorting, but only picked out the best child and returned that first, and left the rest of the array unsorted. Adding insertion sort to sort the whole array of children made the whole search more than twice as fast. I replaced insertion sort with heap sort sometime later, after trying Arrays.sort() and concluding that it is *really* slow. A few tests showed that the sorting algorithm can have quite an impact on the execution time, though; most likely due to the fact that there are many ways to sort children with the same move ordering score (and this occurs alot). Therefore, using different sorting algorithm can easily lead to different speed-ups as well. This effect is becoming less important as more boards are tested, but in single cases, it can make a lot of difference.

Transposition table

An interesting fact about Othello boards is that they can be rotated and mirrored without fundamentally changing anything. Four 90^o rotations combined with one diagonal-mirroring equals eight boards that represent the same actual game situation. On a sidenote, the four moves that black can do from the initial board position are essentially the same because of this, meaning that black's first move is forced – this is said to give white the necessary advantage to win at least 6x6 Othello with perfect play [4, 5]. The outcome of perfect play for 8x8 Othello is as of yet unknown.

This symmetry could in theory result in a large advantage for the transposition table (TT): if you can implicitly store eight boards instead of one per TT entry, you have effectively multiplied your TT size with eight. But in practice, the chances that two symmetric boards are found within a search, are not that high. The performance penalty of bit-storing the board in a way that actually allows for the recognition of symmetric boards is enormous: first, the boards would have to be stored in a way similar to the one shown in the picture (each 16 consecutive positions being stored in one 32-bit integer), then the four integers have to be sorted numerically (with the possibility of two or three of these 'quads' being equal); the same has to be done for the mirrored version of the board, and then the best (numerically largest) of these two sets of four integers has to be taken. This will result in a unique value that is equal for all symmetric boards, but producing it is an extremely costly operation. After testing it a bit, I decided to completely do away with this, and not exploit the symmetry characteristic of boards at all.

0	4	8	12	19	18	17	16
1	5	9	13	23	22	21	20
2	6	10	14	27	26	25	24
3	7	11	15	31	30	29	28
60	61	62	63	47	43	39	35
56	57	58	59	46	42	38	34
52	53	54	55	45	41	37	33
48	49	50	51	44	40	36	32

Transposition tables are limited in size, and therefore require a hashing function to be used on the boards. As a single entry in the TT can cause a large reduction in boards revisited, this hashing function better make sure that the TT is used optimally! After trying about a dozen of hash functions, I found a hash function where about 50% of a number of boards generated in my tests ended up in a free spot (where the number of boards was much smaller than 2^{20}), which was much better than the others I tested, but my opinion still only barely acceptable. I would have liked to try many more possible hash functions, but in order to obtain a reliable result, a huge number of boards has to be hashed with each possible hash function, taking up a lot of time. Reducing 128 bits to 20 bits in a way that produces a unique hash value for boards that are mostly relatively close to eachother (which is obviously desirable for this kind of search) is apparently not easy!

As for the rest of the transposition table: as it is a tightly connected part of the MTD(f) implementation for Awari, I've simply reused most of the code there too; of course I've taken the version that uses a series of flat arrays as that one is *much* faster than the version with an array of TT entry objects. The fact that each board must be represented by four integers, instead of only one for Awari, means that the transposition table can't be as big as Awari's (a factor two smaller to be precise), despite the fact that Othello has a higher branching factor – 10 for Othello vs 3.5 for Awari [6, 9]. Also, no assumption can be made beforehand on the number of integers required to store a board (in principle the code supports board sizes other than 8x8), so

The parallel versions

Parallelization

The differences between the sequential version "seq" and the parallel version "par" of the program is minimal, and are limited to the MTD(f) code that I reused, so no effort was involved here. I don't think that any further optimizations can be made here. One thing to note is the logical but non-obvious fact that the parallel version has to spawn its children in the opposite order that the sequential version visits them, as Satin's job queue has to be implemented as a stack.

In order to limit the number of stolen trivial jobs (where the cost of stealing them exceeds the gain from parallel execution), the parallel version has a depth threshold – if the remaining search depth exceeds this threshold, the remaining jobs are executed sequentially. MTD(f)/Awari does this, but includes a comment saying that experimentation with the threshold value is advisable. I have indeed experimented with this, and the differences between execution times for different values indeed makes it worth experimenting with it (see later on).

The parallelized version of the program can also be improved by transposition table replication. I wanted to stick to Ibis-only features, so I haven't even looked at the sockets version of MTD(f)/Awari. Instead, I have modified the Ibis/RMI version into "par_rmi". This version basically does a sequential broadcast of a TT update after each

10,000 new TT entries. I have tried a few other values instead of the 10,000, but none of these resulted in better overall execution times. Obviously if this value is too low, it will take too much time sending all the broadcasts; if it is too high, it will take too much memory and updates may arrive too late.

Then, I've written a "par_tuple" version that uses the Satin tuple space as replacement for the local or manually replicated transposition table. Noting that the communication overhead was now exceeding the useful application time even for two processors, I added a maximum depth forced on the TT entries (i.e. only all nodes with a distance up to for example 9 from the root would be replicated). This had a positive effect but the communication overhead was still high, especially in the first few iterations of the algorithm. My next attempt, which is also used in the 'tuplespace' implementation of MTD(f)/Awari (which I did not reuse), used the already existing 'depth' variable which is basically the depth left to go; imposing a minimum 'depth' for replication. This reduced the communication overhead to a point where the tuplespace version was actually becoming a candidate at all.

Finally, I've made an attempt at writing a GMI version. More about that below.

Trouble with Satin

During testing of the "par" version, I experienced some weird crashes (out-of-bounds exceptions), although they only seemed to occur when it was run on 16 processors or more, and even then only in some of the runs. Basically, the Mtdf class crashed trying to access a child with an index number that was higher than the number of children in the array. This really puzzled me, given the simplicity of the code that was involved in parallelizing the program.

I've ultimately been able to find out what's going wrong: the fact that the problem did not occur at all when I set the BEST_FIRST option to 'false', was a first hint; adding sequence numbers to depthFirstSearch calls and Done exceptions, and seeing that they sometimes mismatched in the inlet, suggested that the problem was not in my code. And as it turned out, it was indeed a bug in Satin.

I've found out that the bug has already been fixed in Satin after the release of Ibis-1.1 (that I based my work on), so I can leave out the full technical story explaining what went wrong exactly. What it comes down to, is the following (I define a job as an invocation of a Satin-enabled method, here): if a job is started sequentially (i.e. with a normal method call, not through Satin), its parent/grandparent/etc could be aborted, so that it itself is being aborted too, causing it to "return null;" due to SatinC-inserted code. Since the job was not started through Satin, it simply returns to its equally aborted parent. This aborted parent job is then able to spawn up to one child job before detecting that it's aborted as well. That child job, which itself is *not* marked as aborted, could get stolen and sometime later (through inlet execution) have its result delivered in the context of another job.

After finding this out (we're talking about several long days here – I had no prior knowledge about Satin's inner workings – but it was a nice exercise!), writing a patch for the problem was rather easy: simply adding a check in Satin's spawn() code to make sure that no jobs with aborted parents were added to the jobs list, so they could not be stolen either. After switching to my own Ibis build with this patch in place, I never saw the crashes again.

However, I believe that the fundamental problem here is not just this bug. I suspect that this check wasn't there in the first place because it could be assumed that the situation would never occur: an aborted job should never have been able to spawn another job! This, though, is an inherent problem in Satin: if a normally (Java-style) invoked method gets aborted by Satin-inserted code and returns, its (equally aborted) caller is simply left running. All kinds of scenarios can be created where this problem results in things the programmer didn't, and in my opinion shouldn't have to, expect. Besides suffering from null pointer exceptions, the calling method could for example also modify a transposition table based on incorrect results from its aborted children. Or simply do a whole lot of unnecessary extra sequential work.

The issue really lies with the SatinC-inserted "if ((…).aborted) return null;" statements, and there is no trivial way to patch this. Ideally the whole invocation of application code (from the invocation by Satin, to the point where it finds out it's aborted) would be popped from the call stack instead of merely the current method, so I would imagine that one way to solve this problem is to replace the inserted "return null;" statements with statements that throw a Satin-internal exception, which is caught and handled by the Satin code that invoked the (parent) method. As I see it, this fixes the problem in an overall much cleaner way, with the downside that the Satin-exception could possibly be caught by a general exception handler in the application. But this would be something that is much easier for the programmer to understand and deal with.

Since this solution involves quite a few parts of Satin to be (slightly) rewritten, and I am by no means a Satin expert after messing with the code for a few days, I have not taken a shot at actually trying this solution – so I currently cannot prove that this is going to work properly. If it is not a solution, and no other solution can be found for it, then the only reasonable thing that could be done here, is informing the programmer that he/she should be aware of such Satin-specific quirks.

Trouble with GMI

As mentioned, I've also attempted to create a version of the program that uses GMI instead of RMI for the TT replication. After all, GMI promises to overcome the problem of RMI's inherent synchrony (which is indeed costly as I will show later) and offers one-to-many communication in a simple, straightforward way. Combining Satin with GMI therefore seemed logical, but the issues that arose while trying this, forced me to give up in the end.

First of all, GMI and Satin seem to be somewhat incompatible with eachother: if GMI is initialized first, and Satin after that, Satin will hang in its initialization phase. On the other hand, if Satin is loaded first, GMI has no problem loading after that. The problem with that is that GMI group joining acts like a barrier, meaning that all members have to do it at the same logical moment – due to the unpredictability of job stealing of Satin, this is not possible from the satin-spawnable methods. Using static{} code is not going to work either, as Satin gets loaded in the SatinC-inserted main(), which is always invoked after any static{} block in the same class. All the other classes' static{} blocks are only executed upon class reference, which is once again unpredictable as it depends on job stealing.

I ended up hacking SatinC to call my own "postSatinInit()" method for both the main and the child processes, right after the Satin initialization. Loading GMI from that method indeed worked. I have not looked into the reason why Satin fails to initialize after GMI though, so there is probably a much cleaner solution for this.

Another issue with GMI was that it horribly failed generating stub/skeleton code for method invocations that use more than one array of the same type (in this case, two *int[]*s); this was easily fixed by using a wrapper class to store the parameters in, but might be something to look into.

A more serious problem occurred when I was actually testing my GMI code: used on eight processors or more, all instances would deadlock while performing a GroupInvocation, even though there where no 'synchronized' method calls in my code that could possibly cause this. Hoping that it had something to do with the Satin/GMI combination, I created a GMI-only test version that only implemented the core communication part. With that version, I could reproduce the problem on my own system as well, with as little as two instances running.

Basically, if the two instances are set to both perform GroupInvocations (with DiscardReply) repeatedly, with a large array (10,000 ints) as parameter to the call, then they always deadlock after a while – sometimes almost immediately. A stack trace of all threads at that moment (obtained by sending a SIGQUIT signal to java) shows that one upcall is in progress, and that this upcall is blocked reading from its stream. Another upcall is blocked (as only one upcall can be executed concurrently), which explains why netstat shows that some of the sockets' buffers are filled up completely at that moment. The main program thread is blocked writing to a stream.

As I see it, an upcall at node A is only generated when node B has started sending something to the stream associated with the upcall. Also, if node B has started sending to this stream, it will continue doing so until it's done sending. Therefore, A should never have a reason to block receiving in the upcall for a long time – regardless of the state of all the other streams. Unfortunately, the level of abstraction that Ibis imposes on the transport implementations, makes it impossible (for me, at least) to find out which unix socket is associated with a certain read or write operation, so I have been unable to figure out what exactly is going on here. As all the other alternatives I've tried (e.g. with repeated SingleInvocations, even with sending SingleInvocations to only one node) led to the same deadlock situation as well, I had to give up on GMI altogether.

Test conditions

On to the stuff that actually did work, heh. All of the tests below have been performed with a patched build of Ibis 1.1, which fixes the crashing problem indicated above. The tests have all been run on fs0 of the DAS2 cluster, and started with 'grun', which provides a convenient way of turning the remote execution of an Ibis

application into an (admittedly long) one-liner. All of the tests have been performed from the initial Othello position, searching with maximum depth 17.

I realize that to properly evaluate an application like this, a (somehow) representative set of boards should be used; however, this would be mainly important for the evaluation and move ordering functions, and to a much lesser extent for the parallelization of the program, which is why I have decided not to run a full set of tests with lots of other boards. A few odd tests seem to confirm that although the absolute running times differ much from starting board to starting board (requiring the maximum depth for some mid-game boards to be lowered to 13, due to higher branching factor), the relative speed-ups are comparable to those of the tests from the normal starting board.

The tests have all been performed using the Ibis TCP stack. At first, I had a lot of trouble getting Ibis to make use of Myrinet (certain libraries could not be found while they were really there, etc) so I postponed experimenting further with this until later; at the time that I was ready to repeat the tests on Myrinet, the Myrinet switch went down! At that moment I decided to abandon the idea of using Myrinet altogether, and instead stick with Ibis/TCP. I do not know how much of a difference this has made; however, this application is reasonably low-bandwidth and not that latency-dependent compared to other parallel applications, I don't expect the difference to be really big.

Of course, all of the values below are averages taken over a number of runs (usually three or four).

The threshold value

The first question that I wanted to have answered, was about the threshold of the MTD(f) algorithm: from which depth left to go should the algorithm start to do the rest of the work sequentially. Such a threshold can limit the number of stolen trivial jobs, but decreases the number of jobs up for stealing. As this becomes more of an issue as the number of processors grows, I've tested it with a large number of processors. The following values are obtained by running "par_rmi" on 64 processors. I have included odd (as opposed to even) threshold values only, as the differences between successive depths are not really big enough to be meaningful.

The straightforward parallel version "par":

Threshold	**Average time per machine (sec)**			**Steal success rate**	**Bytes transferred**
	App.time	*Par.overhead*	*Sum*		
1	633.558	62.488	696.045	49.246 %	459,896,384
3	478.850	31.281	510.130	39.797 %	212,576,221
5	443.272	27.466	470.738	14.352 %	111,570,581
7	424.481	75.335	499.816	1.051 %	164,001,750
9	397.272	336.010	733.282	0.040 %	691,139,981

The parallel version with RMI replication "par_rmi":

Threshold	**Average time per machine (sec)**			**Steal success rate**	**Bytes transferred**
	App.time	*Par.overhead*	*Sum*		
1	339.175	114.449	453.624	15.642 %	446,298,293
3	276.978	90.966	367.943	8.355 %	272,246,072
5	230.545	78.449	308.994	2.912 %	172,727,560
7	212.477	163.820	376.297	0.245 %	314,371,865
9	244.581	367.681	612.262	0.024 %	726,614,088

The parallel version with tuplespace replication "par_tuple":

Threshold	**Average time per machine (sec)**			**Steal success rate**	**Bytes transferred**
	App.time	*Par.overhead*	*Sum*		
1	269.516	82.675	352.191	31.818 %	433,432,644
3	187.443	50.319	237.761	21.019 %	205,049,945
5	185.857	41.468	227.325	7.255 %	105,308,140
7	191.578	92.446	284.024	0.590 %	161,531,206
9	181.916	311.748	493.665	0.022 %	617,847,126

The first observation here is that the lowest threshold does not lead to the best results, showing just how useful it is to apply such a threshold at all. A high threshold is not helping either, as the number of spawned jobs drops to a point where stealing a job becomes extremely difficult, resulting in high idle times; this naturally also decreases the average useful application time per machine, even though the amount of work performed is not at all lower than for the other runs.

In all three cases, a threshold of 5 resulted in the best execution times, least parallel overhead and least bytes transferred, so this is the value that I have used in the remaining tests.

The "par_tuple" replication depth

The next question was which replication depth was to be used for the tuplespace "par_tuple" version. Clearly there is a balance here too, with the amount of traffic overhead versus the gain from cut-offs due to replicated entries. Unfortunately, at the time that I was testing these values, someone else was running a large number of jobs of several hours each on 32 processors, so I had to resort to testing on 32 processors instead of 64. I don't expect the relative differences on 32 and 64 processors runs to be that big here, anyway.

Min. depth	**Average time per machine (sec)**			**TT sorts+hits**	**Total tuple space broadcast**	
	App.time	*Par.overhead*	*Sum*		*Time (sec)*	*Messages*
5	320.205	47.048	367.254	877	234.940	258,744
7	301.657	30.187	331.844	275	50.573	41,372
9	357.053	26.243	383.296	109	21.006	5,443
11	534.848	27.121	561.969	33	15.437	805

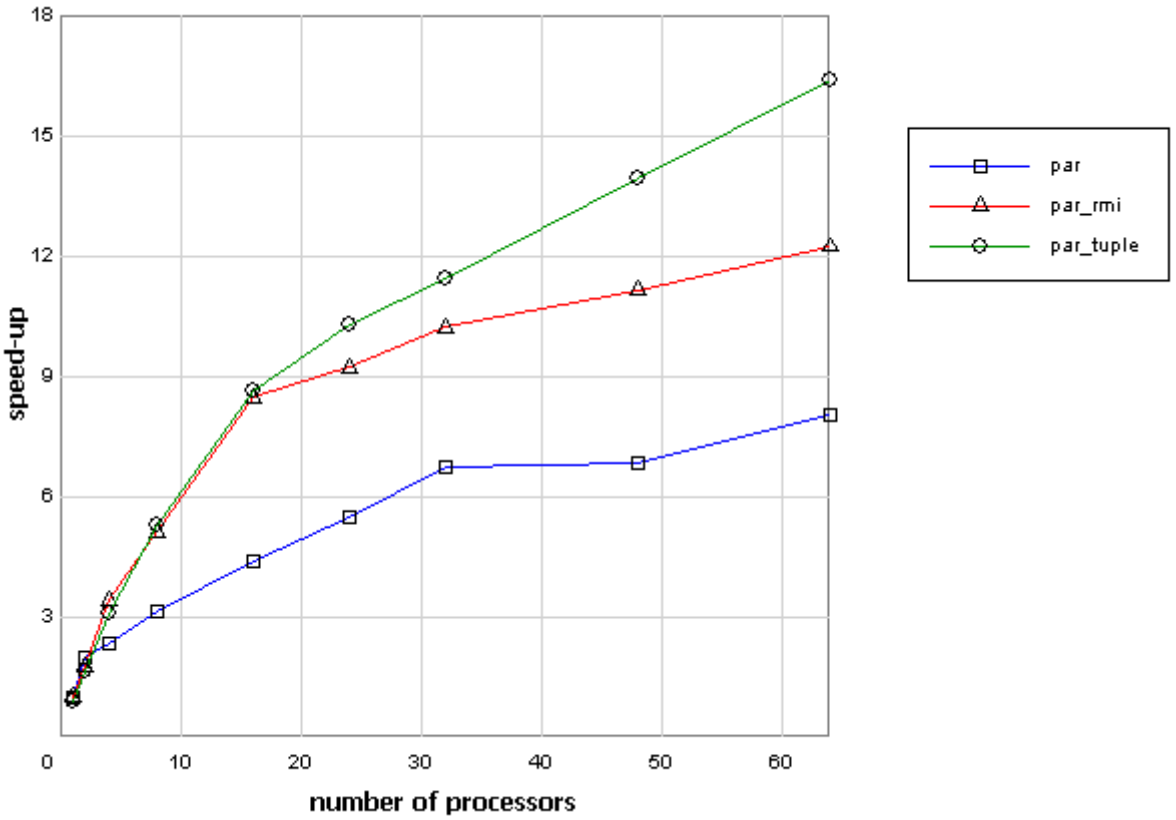
Despite the fact that "luck" with respect to aborted jobs and replicated TT entries plays a huge role, these averages are the outcome of a sufficiently large amount of tests that this factor can be considered eliminated. The table indeed shows that there is a balance between communication overhead (which is the tuple broadcast time – the lower the better) and the number of TT sorts/hits (the higher the better). It is remarkable how few TT sorts/hits can result in so much improvement in elapsed application time, especially considering that several millions of boards are visited during the run; it shows that one TT hit can lead to a large amout of subjobs being skipped. The minimal depth resulting in the lowest average execution time in this specific case is 7, which is what I've decided to use in the speed-up tests as well.

Speed-ups

With the best values obtained in the tests above, speed-up tests yielded the following results.

# procs	**seq**	**par**			**par_rmi**			**par_tuple**		
	(3)	*(1)*	*(2)*	*(3)*	*(1)*	*(2)*	*(3)*	*(1)*	*(2)*	*(3)*
1	3776	3839	0	3839	3791	0	3791	4255	0	4255
2	-	1925	2	1927	2135	2	2137	2326	5	2331
4	-	1624	4	1626	1104	5	1109	1227	6	1233
8	-	1204	12	1216	730	10	740	704	10	715
16	-	853	13	866	432	15	447	418	20	438
24	-	675	18	693	382	28	410	346	22	368
32	-	542	20	562	334	36	370	301	30	331
48	-	528	24	552	273	66	339	236	34	271
64	-	442	29	471	236	73	309	189	47	231

(1) average useful application time per machine (seconds)
(2) average parallel overhead time per machine (seconds)
(3) total running time (seconds)



Discussion

The first observation to make is that the parallel version “par” is almost not at all slower than the sequential version “seq”. With a difference of about 1%, this is much better than the 4.9% reported for MTD(f)/Awari [7], and can be attributed to the fact that the computations on boards for Othello are by nature much more intensive than those on boards for Awari. This, and the fact that the applied threshold makes sure that large parts are executed sequentially anyway, make the parallel overhead for 1-processor runs almost negligible. Impressive! The same goes for the TT-replicating “par_rmi”, despite the fact that it is a little slower due to the fact that it does some useless work maintaining two copies of the same last 10,000 TT entries. The “par_tuple” tuplespace version is much (close to 12%) slower, which is largely due to the fact that it can not use flat transposition table arrays as the rest does, meaning it has to perform memory allocation operations for (nearly) every new TT entry. It also suffers from the fact that tuple broadcasts take a short while to arrive locally.

However, the most interesting fact is that the “par_tuple” version is by far the fastest on 64 processors: it is over 16 times as fast as the sequential run. The tuplespace version is considerably faster than “par_rmi” on 64 processors (which achieves a speed-up of about 12), even though they are about as fast on up to 16 processors.

Taking a closer look at the TT statistics from the runs, the results are not really that surprising. Overall, a hit/lookup ratio (hits including children sorts) of about 1 to 2 percent for full TT replication as “par_rmi” does, is not unusual; this appears to be consistent with what MTD(f)/Awari gets. However, many of these hits are for close-to-leaf nodes, meaning that the gain is low. In fact, only about 0.1% of all lookups seem to result in a hit that leads to substantial improvement, and these hits are the ones for nodes closer to the root node. Therefore, only replicating those will lead to approximately the same computational advantage, while eliminating the overhead involved in replicating the rest. Since this is exactly what “par_tuple” does, this version achieves the best results. On average, it spends only about 2 seconds per machine on tuplespace broadcasts, whereas “par_rmi” spends about 45 seconds per machine doing its broadcasts, although these are (on one hand) performed less frequently and (on the other) synchronous.

For the same reasons, I would expect a version that uses GMI to do group-broadcasting of the best entries (the same that “par_tuple” stores) while maintaining its own full local copy using flat arrays, to achieve even better results, although possibly only very marginally. Unfortunately I cannot prove this.

Needless to say, the “par” version that does no TT replication, doesn’t achieve the same speed-ups, although still a factor of eight on 64 processors compared to the sequential run. That all measured speed-ups are higher than those measured for MTD(f)/Awari [7], was to be expected: as mentioned before, both the the job granularity and branching factor of Othello are higher than those of Awari.

So why aren’t the speed-ups even higher? For the “par_tuple” version, the 64-processor run searches close to three times as many boards as the sequential run (160 million vs 55 million), i.e. there is a search overhead factor of three; this indeed more or less matches the factor 20 ‘pseudo-speed-up’ of *application time* compared to the sequential run. The search overhead could attributed to a combination of non-replicated TT entries, propagation time of replicated TT entries and propagation time of abort messages, but as shown before, replicating more TT entries does not improve things, and the other two factors I have no influence on.

The *parallel overhead time* that makes up the rest of the execution time, consists almost entirely of job stealing time, and I’m afraid that not much can be improved there: as I’ve shown before, if the threshold value is lowered, more steal attempts will succeed, but the overall execution time will still be higher due to the fact that more boards are visited because of non-replicated TT entries; once again, if the replication depth is lowered to overcome this, the tuple broadcast and (de)serialization time together exceed the time gained from a higher job steal success rate. There really is a balance of balances here, and I believe to have come close to the optimal values for it.

Still, there is room for quite some improvement. The strength of the MTD(f) algorithm relies on good evaluation and move ordering functions, and the evaluation function can of course be improved, although this would take a person who has more experience with Othello than I have. Changing the evaluation function would also quite possibly require the move ordering function to be changed; I would imagine that like the evaluation function, the move ordering might also gain from a system where the importance of its factors depends on how far the game has progressed. Right now, all other variables are also fine-tuned only for runs from the starting position, testing with a much higher number of starting boards should result in better tuned values for all situations as well.

Conclusion

Overall, I am rather satisfied with the project and its results. Even though a large part of the actual code is not mine, I have a *much* better understanding of how Satin and other Ibis parts work, how Othello works and what it takes to write a search application like this, to parallelize it and to actually test it on a parallel system. I must add that given the general lack of documentation for Ibis components, I regularily had to check the source code to see what I could expect (obviously the mentioned problems I ran into, required this as well), which once again reinforces my opinion that the high level of systems like Satin is never purely beneficial to the programmer, especially not when problems arise. That said, Satin and other Ibis components are very easy to use, and simply eliminate the need to spend lots of time on low-level parallelization mechanisms at very low cost, which is definitely applaudable!

Regarding Othello – how feasible is it to beat the game, i.e. compute the best play from start to end? The depth of 17 that I tested with, is *nowhere near* the depth of 64 that the average game has, especially considering that the algorithm is getting slower at each added depth. The achieved speed-ups are not quite good enough either. All in all, the program requires a much better evaluation function, and quite possibly a different language and parallelization base in order to even come close to this. Although slightly disappointing, this is not entirely unexpected – if it were this simple to beat Othello, it would already have been done by now.

References

[1] Andersson, G. *Writing an Othello program*. 2004. http://www.radagast.se/othello/howto.html
[2] Buro, M. *An Overview of NECI's Generic Game Server*. http://www.cs.unimaas.nl/olympiad/workshop/Buro.ppt
[3] Buro, M. *Logistello.* 2003. http://www.cs.ualberta.ca/~mburo/log.html
[4] Feinstein, J. *The perfect play line in 6x6 Othello*. 1993. http://www.maths.nott.ac.uk/personal/jff/othello/6x6sol.html
[5] Feinstein, J. Newspost: *Re: 6x6 Othello*. 1993. http://www.ics.uci.edu/~eppstein/cgt/othello.html

[6] Kishimoto, A., Schaeffer, J. *Transposition Table Driven Work Scheduling in Distributed Game-Tree Search*. 2002. http://www.cs.ualberta.ca/~jonathan/Grad/Papers/ai02.ps
[7] Nieuwpoort, R. van. *Efficient Java-Centric Grid-Computing*. 2003.
[8] Plaat, A. *MTD(f): A Minimax Algorithm faster than NegaScout.* 1997. http://www.cs.vu.nl/~aske/mtdf.html
[9] Plaat, A., Schaeffer, J., Pijls, W., Bruin, A. de. *Nearly Optimal Minimax Tree Search?* 1994. http://www.cs.ualberta.ca/~jonathan/Grad/Papers/TR94-19.ps
[10] *Pressman Toy Instructions for Othello.* http://www.pressmangames.com/instructions/instruct_othello.html
[11] *The Free Dictionary: Encyclopedia article about Reversi.* http://encyclopedia.thefreedictionary.com/Reversi
[12] Voges, S. *Marvin: Yet Another Othello Applet*. 1999. http://litefaden.com/marvin/
[13] Wolf, T. *Inside Reversi/Othello: The Anatomy of a Game Program*. 2000. http://home.tiscalinet.ch/t_wolf/tw/misc/reversi/html/