

Playing Parity Games on the Playstation 3

Jorne Kandziora

j.kandziora@student.utwente.nl

ABSTRACT

The Small Progress Measures parity game algorithm, developed by Marcin Jurdziński, is perfectly suitable for parallel execution. The current parallel algorithm is programmed for the x86 computer architecture. In this paper, a clustering approach is introduced, as well as a variation on the Small Progress Measures algorithm, to enable execution on the IBM Cell Broadband Architecture.

Keywords

Playstation 3, Cell Broadband Engine, Small Progress Measures, Parity Game, Model Checking

1. INTRODUCTION

Model checking, a technique used for evaluating the correctness of software and hardware, relies on a method used in algorithms for playing parity games; games played by two players on a graph. The performance of these algorithms can be optimized and distributed by executing them in parallel. Further optimization might be done by using the new IBM Cell Architecture.

The rest of this section will give an introduction to the Cell BE Architecture, Parity Games and the Small Progress Measures algorithm. Section 2 states the goal of this research. Section 3 lists work related to this research. Section 4 proposes the changes to the Small Progress Measures algorithm and Section 5 states the issues addressed to run this altered algorithm on a Playstation 3. Sections 6, 7 and 8 describe the altered algorithm itself. Finally, Section 9 will state some future research on this topic and Section 10 contains the conclusions of this research.

1.1 Basic Characteristics of the Cell BE Architecture

The Cell Broadband Engine Architecture (Cell BEA) is a product of the cooperation of IBM, Sony and Toshiba [2]. The Sony Playstation 3 contains a Cell Architecture, consisting of a single Power Processor Unit (PPU) and multiple Synergistic Processor Units (SPU), all connected to a shared bus. The SPU's can act as stand-alone processors, but they have limitations: they can only communicate with the Cell hardware through the PPU. The PPU is a 64-bit processor. The SPU's are 32 bit vector processors, which can do multiple (4 or more) operations at the same time and are optimized for floating point operations. The memory of an SPU is isolated from the main memory of a Playstation 3. An

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission.

10th Twente Student Conference on IT, Enschede 23th January, 2009

Copyright 2009, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science

SPU has a local memory (local store, 256 KB), which can interact with the main memory in small blocks using Direct Memory Access (DMA).

SPE or SPU?

Throughout this document, both the terms SPU and SPE, or PPU and PPE are used. The term SPU (Synergistic Processor Unit) denotes the actual processor the program is running on, while the Synergistic Processing Environment (SPE) denotes the environment the SPU is running in: an SPE contains not only the SPU, but also a "memory flow controller", which handles the associated registers and DMA intrinsics.

Similar naming is used for a PPU and a PPE: the SPE program interacts with the associated Power Processing Environment (PPE), which uses a Power Processing Unit (PPU) to do the actual calculations.

1.2 Parity Games

A parity game is a game played by two players on a (directed) graph with numbered (prioritized) vertices. Hartmut Klauck gives a good description of the game and the various algorithms available [5]. In a parity game the vertices are split into vertices of player \square and vertices of player \circ . A starting vertex is declared, after which a token is passed between vertices to mark the progress. The goal of the game is repeatedly reaching the lowest priority in an infinite game. Player \square tries to win by reaching the lowest even priority, while player \circ tries to do the same by reaching the lowest odd priority. In parity games there is always a winner.

For this research we assume that the graph used is finite. Parity game algorithms find the winning set of player \square — the set of vertices for which it is certain that player \square will win the game in a perfect play — and the strategy needed to win. An example of a parity game graph, including the winning set of player \square (in gray) is shown in Figure 1.

1.3 Small Progress Measures

The Small Progress Measures algorithm [3] associates a vector M_G (progress measure) with every vertex in the parity graph, counting the priorities found in the graph. In this algorithm, the priority vectors are compared to each other using their lexicographical ordering. All vectors are initialized to $\vec{0}$. The vector M_G^v extends this lexicographical ordering by taking \top as the biggest element in the ordering.

A progress function Prog is declared to increase these vectors. $\text{Prog}(g, v, w)$ denotes the smallest successor m of w such that $w \geq_{p(v)} m$ if $p(v)$ is even and $w > m$ or $w = m = \top$ if $p(v)$ is odd, where $p(v)$ denotes the priority of the priority of v .

A lifting operation is declared, which will increment the priority

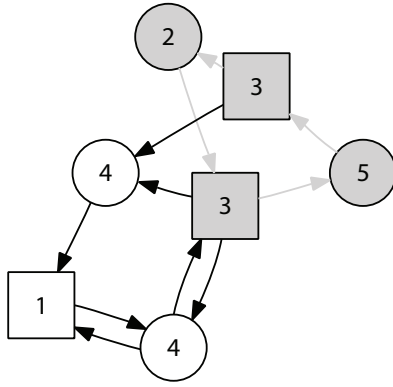


Figure 1: Example parity game graph

vector, if possible, using the direct neighbours of a vertex:

$$\text{Lift}(g, v)(u) = \begin{cases} g(u) & \text{if } u \neq v \\ \max\{g(v), \min_{(v,w) \in E} \text{Prog}(g, v, w)\} & \text{if } u = v \in V_{\square} \\ \max\{g(v), \max_{(v,w) \in E} \text{Prog}(g, v, w)\} & \text{if } u = v \in V_{\circ} \end{cases} \quad (1)$$

The algorithm uses this operation to monotonically increase all vectors, until no vector can be increased. The winning set and winning strategy for player \square then consists of the vectors with an associated \top measure.

2. PROBLEM STATEMENT

The goal of this research is to develop, test and benchmark a modified Small Progress Measures algorithm on a Playstation 3. The current parallel implementation of the Small Progress Measures algorithm is built for a standard multi-core CPU [7]. This research will develop a version of the Small Progress Measures algorithm which is poised for running on the Cell architecture. Developing such an algorithm raises several issues, addressed below.

The parallel version of the Small Progress Measures algorithm, as proposed by Jaco van de Pol and Michael Weber, uses shared memory to contain the graph and solution data [7]. SPE's on the IBM Cell Architecture can only access the main memory via a DMA request through the Element Interconnect Bus. A local copy of a subgraph to work on should therefore be fetched into the SPE's local memory. This can be done without damaging the integrity of the algorithm; Jurdzinski shows that lifting performed on a subgraph yields the same result as lifting performed on the entire graph [3]. Since DMA requests are slow compared to the SPU speed, choosing partitions (clusters) with a reasonable work load is important.

3. RELATED WORK

Jaco van de Pol and Michael Weber show that the Small Progress Measures algorithm can be parallelized and give an implementation of this parallelization [7]. Their research gives several approaches for choosing jobs given to the parallel executions of the algorithm.

Marcin Jurdziński et al. provide an algorithm that uses a deterministic approach to find the winning solution in subexponential time [4].

Software managed threads can boost SPU performance if DMA

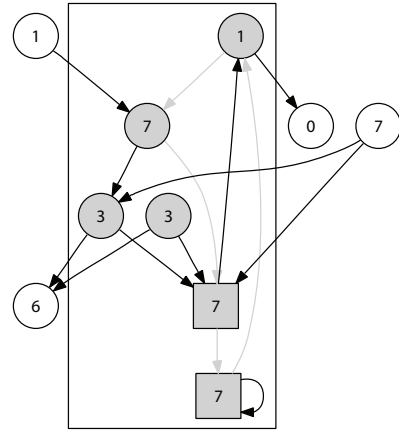


Figure 2: A small parity game cluster with a highlighted cycle

requests are the bottleneck for an algorithm [1]. This approach requires multiple jobs available for one SPU, which will result in partitioning a problem in smaller subgraphs.

4. ALGORITHM MODIFICATIONS

The algorithm and its implementation is closely related to the multi-core implementation made by van de Pol and Weber [7]. Data structures and methods are re-used where possible, to avoid doing the same work twice. The PS3 algorithm however, uses a different parallel execution, to meet the hardware requirements of the PS3.

4.1 Clustering

The parity game graph needs to be partitioned to fit it on the SPUs. Therefore, so-called clusters are declared. The Small Progress Measures algorithm will be executed repeatedly on each of these clusters, until a stable global situation is reached. Ideally, a cluster contains a cycle, which can be lifted to some extent before reaching a stable situation. An example of a small parity game cluster is shown in figure 2

4.2 Reaching a stable global situation

Reaching a stable global situation can be done by counting the number of lifts performed by the clusters. When an entire run through all clusters yields no additional lifts, the algorithm finishes. In pseudocode:

Listing 1: Cluster lifting

```

1 do:
2   lifts := 0
3   for each cluster do
4     lifts := lifts + LiftCluster(C);
5 while(lifts != 0)

```

The LiftCluster operation will perform as much lifts on a cluster as possible. The lifts are performed on the SPU. To ensure that the algorithm yields the right answer, the LiftCluster operation only modifies the progress measures of the cluster it operates on; vertices outside a cluster will only be changed by their own LiftCluster operation.

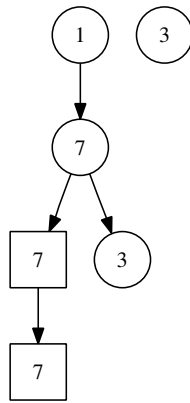


Figure 3: A tree view of figure 2

4.3 Clustering Approach

Multiple clustering strategies can be used for partitioning the graph. This implementation uses a k-bounded depth first search: starting from a certain vertex, the outbound edges are followed to a certain depth. Edges are pre-emptively added to a cluster, until a cluster is full. This method will most likely result in a lot of forward edges and some edges to vertices already included in this cluster. A tree view of figure 2, showing this clustering technique, is shown in figure 3.

Van de Pol & Weber walk backwards through their graph when lifting the measures. This strategy maximizes the number of lifts and exploits a structure with a lot of forward edges [7]. This approach can be used perfectly for the evaluation of individual clusters using the clustering strategy described above.

The implementation also uses the backwards strategy described above to select which cluster will be loaded on an SPU. Using this same strategy will hopefully exploit the forward edges between clusters. A different approach might use a focus list, similar to the focus list used in the parallel implementation [7], to keep track of the clusters with a lot of changes. These clusters can be loaded before working on the rest, to further optimize the lifting of the clusters.

Using other clustering strategies might produce clusters with different characteristics, which will probably benefit from other lifting strategies and cluster selection techniques. In this research, clusters are not allowed to contain the same vertices. Allowing this might result in clusters with more cycles and more efficient strategies. This might, however, change the algorithm significantly. The correctness of the algorithm must therefore be guarded.

5. CELL SPECIFIC IMPLEMENTATION ISSUES

5.1 Loading Programs to the SPU

The PPU is responsible for loading programs to the SPE's. The PPU needs to create an SPE context and then load an SPU program into this context. This context can then be loaded onto an SPE. This last step should be done in a separate thread. The parity game program uses POSIX threads to do this. The PPU thread locks when an SPU program is loaded. A 64-bit value (e.g. a pointer) can be sent to the SPU when the program is loaded. The

parity game program does not use this option. Mailboxes are used to exchange pointers and control blocks.

5.2 Exchanging Pointers Between the SPE and the PPE

The SPU is a 32-bit processor, while the PPU has a 64-bit architecture. Likewise, the PPU and the SPU use a different pointer format. To overcome this issue, a union called `addr64` is used. This union can be used to split the pointer in two 32-bit integers, which can then be exchanged using mailboxes.

5.3 Communication Between SPE's and the PPE

Direct communication between SPE's and the PPE happens via mailboxes and signals. A mailbox is a mechanism for exchanging messages between the SPE's and the PPE. SPE's can put a message in the PPE's mailbox and vice versa. SPE's can also fill each other's mailboxes. A mailbox message is 32 bits long.

There are two interfaces on the SPE to use mailboxes: signaling mailboxes and non-signaling mailboxes. Signaling mailboxes raise an event (interrupt) on the PPE thread and can be used to notify the PPE of events happening on the SPE's, regardless of the order in which they happen.

5.4 Direct Memory Access

The SPE can use DMA to transfer large blocks of data to and from main memory. SPE's need to signal the PPE to request a DMA transfer. The PPE sets up the transfer and the SPE is signaled again when the transfer is done.

Up to 32 transfers can be issued at the same time. A bit mask is used to identify the transfers. This bit mask can be used after the transfer to see which transfers are done or to wait for specific transfers.

Data is transferred in blocks of 32 – 128 bits. Data should be at least 32-bit (preferably 128-bit) aligned. Structures that need to be transferred between the main memory and the SPU should be padded to fit 32-bit blocks.

5.5 Control Blocks

To synchronise communication between the PPU and the SPU's, the algorithm uses so-called control blocks. These structs contain information about the jobs issued to the SPU's. An SPU can use this information to load the data it needs from the main memory and to save the data later on.

5.6 Game Representation

A graph is represented by its vertices. For every vertex, the following information is used:

- The priority of the vertex
- The owner (player) of the vertex
- The outgoing edges (neighbours) of the vertex

To play a game, some additional information is needed. The following global game information is saved:

- The game graph
- Cluster definitions
- A progress measure for every vertex
- The maximum priority encountered in the graph
- A maximum (top) vector for the graph

6. RANDOM GRAPH GENERATION

A random graph can be generated to provide test data for the algorithm. To do this, the PPE seeds the available SPE threads with a random seed. It then assigns jobs to the SPEs until an entire graph is generated.

The PPE assigns jobs in a cyclic matter. Jobs are first assigned to SPE 1–6. the SPEs start working and the PPE waits for all of the SPE's to finish. The PPE then assigns the next 6 jobs to the SPEs. Using this strategy, the algorithm will always produce the same graph when it is seeded with a certain value, as opposed to an event-based strategy, where the SPU's are assigned a new job as soon as they are finished. In pseudocode:

Listing 2: PPU graph generation

```

1 n := 0;
2 used_spus = 0;
3 send_graph_pointers();
4 while(n < graph_size){
5   if(used_spus < available_spus){
6     job_vertices = min(graph_size - n, job_size);
7     generate_vertices(n, job_vertices);
8     n = n + job_vertices;
9     used_spus++;
10  } else {
11    wait_for_spus_to_finish();
12    used_spus = 0;
13  }
14 }
15 wait_for_spus_to_finish();

```

Communication happens using mailboxes. A message is delivered to the SPE, containing a pointer to a control block. The SPE uses this block to get the data it needs to generate. Lines 3, 7 and 15 send mailbox messages to the SPU..

The SPE then starts generating random outgoing edges, a random priority and a random player for the vertices. Once all data has been generated, the data is saved back to the main memory and the SPE signals the PPE that it has finished. The PPU waits for this signal and assigns the next job. When the entire graph is generated, the PPU tells the SPUs that they may finish. In pseudocode:

Listing 3: SPU graph generation

```

1 graph = load_graph_content(get_graph_pointer);
2 while({n, job_vertices} = get_job()){
3   vertices = load_vertices(n, job_vertices);
4   for(vertex : vertices){
5     vertex.priority = random(max_priority);
6     vertex.player = random(0,1);
7     for(edge : vertex.edges){
8       edge = random(count(vertices));
9     }
10  save_edges(vertex);
11  }
12  save_vertices(vertices);
13  notify_ppu();
14 }

```

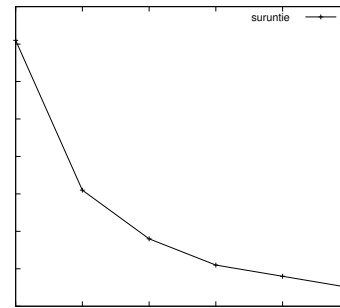


Figure 4: Times measured for random graph generation using multiple cores

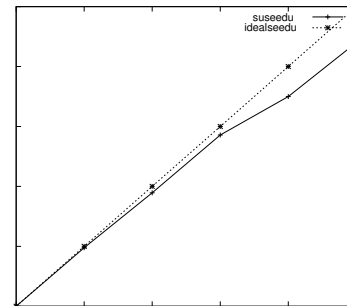


Figure 5: SPU speedup

Lines 1, 3, 10 and 12 use DMA transfers to transfer data from and to main memory. Lines 1, 3 and 13 use mailbox messages to communicate with the PPU.

6.1 Benchmarking the PS3 Speed

To measure the overhead of multiple SPUs working in parallel, a simple benchmark is performed on 1 to 6 SPUs. The goal of this experiment is to measure the penalty of multiple threads working in parallel, compared to a single thread doing the same job.

The benchmark is performed by repeatedly generating a graph of half a million vertices, with all vertices having 20 outgoing edges. The outgoing edges are generated on 1 to 6 SPU's. To get some measurable times, the graph is generated 20 times. The measured times are wall clock times. figure 4 shows the measured results. figure 5 shows the associated speedup of this algorithm.

The algorithm used for this benchmark uses DMA transfers. A decrease of the speedup can be seen in figure 5 when using 4 or more SPUs. This decrease can be caused by a busy Element Interconnect Bus or by a busy PPU, which cannot keep up with the speed of the SPE's requesting DMA transfers. In both cases, DMA transfers are most likely the bottleneck of this algorithm.

This theory can be tested by performing the same benchmark without actually sending the data to the main memory. If approximately the same fallback is measured, DMA is not the bottleneck of this algorithm and future research is needed to determine the source.

If DMA is the bottleneck, a similar decrease will probably be seen when benchmarking the actual parity game algorithm. If

this algorithm is less DMA intensive, the decrease will be less. If, however, the algorithm uses more DMA transfers, the decrease will be greater.

The measured decrease can also be caused by the cyclic assignment of jobs: some SPEs might already be waiting for a new job, while the others are still generating vertices. To test this, an event driven approach can be used to assign a new job to the SPE as soon as it finishes. If the decrease is smaller, then this is a source of this problem. The actual parity game algorithm already uses an event driven SPU job assignment. If cyclic assignment is indeed the cause of the decrease, the result will probably be smaller for this algorithm.

7. GRAPH CLUSTERING

To meet the memory restrictions of an SPU, the algorithm needs to be partitioned into clusters that fit on an SPU. An SPU can then load such a cluster, work on it for a while and write the result back to main memory. The PPE maintains the global state of the graph.

There are multiple approaches for this “clustering”. In an ideal situation, a cluster contains a cycle. An SPU can work on such a cluster for a while before reaching a stable local state and reporting back to the PPE.

Clustering is done on the PPU, using the k-bounded depth first search described above, which will hopefully isolate some cycles on a cluster. The clustering implementation uses a stack to remove recursion when performing the search. In pseudocode:

Listing 4: graph clustering algorithm

```

1 generate_cluster(size, depth){
2   list cluster;
3   stack worklist;
4   while(cluster.size() < size){
5     vertex = get_initial_free_vertex()
6     cluster.add(vertex);
7     worklist.push(vertex);
8     while(!worklist.empty())
9       && cluster.size() < size){
10      vertex = worklist.peek();
11      if(worklist.size() >= depth){
12        worklist.pop();
13      } else {
14        vertex = get_next_free_child(vertex);
15        if(vertex != null){
16          worklist.push(vertex);
17          cluster.add(vertex);
18        } else {
19          worklist.pop();
20        }
21      }
22 }

```

Figure 2 shows a very small cluster (6 vertices with a depth of 3) generated by this algorithm. SPU memory limitations allow clusters with a size of up to approximately 1500 vertices to be loaded on an SPU.

The clustering implementation generates clusters without overlap. The algorithm might also yield the right result when clusters have overlap, but no research is done in this direction.

8. THE LIFTING ALGORITHM

8.1 The PPU Algorithm: Managing Clusters

The PPU uses the lifting approach for clusters, introduced in Listing 1, to lift the clusters. The PPE provides all SPE's with a cluster to work on. The SPEs report back to the PPE using a signal in mailbox message when they are done working on a cluster. This message contains the amount of lifts performed by an SPE.

The PPE then provides the SPE with a new job. If an SPE finishes with at least one lift performed, the PPE will mark the game graph dirty: The algorithm will need at least one more run to reach a stable situation. In pseudocode:

Listing 5: cluster lifting algorithm

```

1 dirty = true;
2 used_spus = 0;
3 while(dirty){
4   mark_all_clusters_dirty();
5   dirty = false;
6   while(!all_clusters_done()){
7     while(used_spus < available_spus){
8       cluster = get_cluster_to_work_on();
9       mark_cluster_clean(cluster);
10      send_cluster_to_spe(cluster);
11      used_spus++;
12    }
13    clusters = get_response_from_spes();
14    for(cluster : clusters){
15      if(get_lifts_performed(cluster) > 0)
16        dirty = true;
17      used_spus--;
18    }
19  }
20 }

```

Line 13 in the algorithm above locks to avoid busy waiting. The algorithm continues until all clusters reach a stable situation (no more vertices are lifted). If an entire run through all clusters yields no additional lifts, the graph has reached a stable situation and the algorithm is finished. The `get_cluster_to_work_on()` function will provide the clusters in backwards order: the last cluster generated will be returned first and the first cluster generated will be return last.

8.2 The SPU Algorithm: Lifting

The SPU program performs the actual lifting of the vertices. The SPU program first loads a local copy of a cluster into memory: cluster vertices, edges and measures are loaded. It then starts lifting the vertices in this cluster using the lift operation.

The structure of the lifting is similar to the PPU program. First, the entire cluster is marked dirty. Then the algorithm marks the vertices dirty and starts lifting them one by one. After a complete run through the vertices, the cluster might be dirty again and the process is repeated. If a cluster is still clean, the algorithm finishes. The entire algorithm in pseudocode:

Listing 6: vertex lifting algorithm

```

1 dirty = true;
2 lifts = 0;
3 while(dirty){
4   mark_all_vertices_dirty();
5   dirty = false;
6   while(!all_vertices_done()){
7     vertex = get_vertex_to_work_on();

```

```

8   mark_vertex_clean(vertex);
9   measure_to_lift = get_measure_copy(vertex);
10  for(edge : neighbours){
11      if(is_external(edge))
12          measure_to_compare =
13              Prog(fetch_external_measure(edge));
14      else
15          measure_to_compare =
16              Prog(fetch_local_measure(edge));
17
18      if(vertex.player == circle)
19          measure_to_lift =
20              max(measure_to_lift, measure_to_compare);
21      else
22          measure_to_lift =
23              min(measure_to_lift, measure_to_compare);
24  }
25
26  if(measure_to_lift != get_measure(vertex)){
27      save_measure(vertex, measure_to_lift);
28      lifts++;
29      dirty = true;
30  }
31 }
32 }
33 return lifts;

```

The *Prog()* function in this implementation is the function described in section 1.3.

The direct neighbours of a vertex, used for the lifting operation itself, are split into two types of vertices:

- Neighbours within the cluster (internal vertices)
- Neighbours outside the cluster (external vertices)

The measures of internal vertices are already loaded in main memory. Performing the lifting operation, described in section 1.3, is directly possible for these vertices. The measures of external vertices, however, need to be fetched from main memory.

This implementation fetches the external neighbours every time they are used. Cache strategies might be used to keep frequently used measures in a local SPU cache, but these strategies are not explored in this research.

9. FUTURE RESEARCH

The algorithm described in this paper is not yet benchmarked. Although the implementation shows that execution of the algorithm on the IBM Cell Architecture is possible, the implementation should be compared to other available parity game algorithms. The speedup penalty, described in section 6.1 should be measured for this execution to see how setups similar to the Playstation 3 perform when execution is divided amongst more processors.

In our experiments only one clustering technique was considered: a k-bounded depth first search. The clusters made by this technique did not have overlapping vertices. Other clustering techniques are a logical extension of this research. These techniques might enable execution of the algorithm on other parallel systems, such as graphical processors [6].

The clustering approach divides the graph in several clusters, but the extensive need of external neighbours still blocks a true distributed execution of the clusters. Developing cache mechanisms

for these neighbours can enable the use of the described clustering approach in distributed systems.

Finally, only one lifting approach was considered: lifting the vertices backwards, compared to the ordering made by the clustering algorithm. Other lifting strategies can be considered to speed up the algorithm, such as the focus list approach proposed by Van de Pol & Weber. [7]

10. CONCLUSIONS

The Parallel Progress Measures algorithm is suitable for execution on the IBM Cell BE Architecture. A clustering of a parity game is proposed to allow execution of the algorithm, without knowledge of a global state. This clustering enables not only parallel, but possibly also distributed execution.

More research is needed to draw any conclusion on the actual performance of this approach. The approach itself, however, opens doors to distributed executions of model checking applications. We hope that heterogenous environments will be considered more when tackling these problems.

Acknowledgments

I would like to thank Michael Weber for his valuable insights and guidance throughout the project and for providing the parallel implementation of the Small Progress Measures algorithm as a reference.

I would like to thank Roel Baardman for his help with the specific Cell BE intrinsics and for providing a Playstation 3 to run this project on.

REFERENCES

- [1] David A. Bader, Virat Agarwal, Kamesh Madduri, and Seunghwa Kang. High performance combinatorial algorithm design on the cell broadband engine processor. *Parallel Comput.*, 33(10-11):720–740, 2007.
- [2] IBM. Cell broadband engine architecture. technical report version 1.02. Technical report, IBM Systems and Technology Group, 2007.
- [3] Marcin Jurdzinski. Small progress measures for solving parity games. In *STACS '00: Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301, London, UK, 2000. Springer-Verlag.
- [4] Marcin Jurdziński, Mike Paterson, and Uri Zwick. A deterministic subexponential algorithm for solving parity games. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 117–123, New York, NY, USA, 2006. ACM.
- [5] Hartmut Klauck. Algorithms for parity games. *Automata logics, and infinite games: a guide to current research*, pages 107–129, 2002.
- [6] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM.
- [7] Jaco van de Pol and Michael Weber. A multi-core solver for parity games. *Electron. Notes Theor. Comput. Sci.*, 220(2):19–34, 2008.