Optimizing Job Scheduling on Multicore Computers

Natalya Bondarenko

Erasmus Mundus Master in Complex Systems Science,
Centre for Complexity Science, University of Warwick

Supervisor: Dr. Ligang He
Department of Computer Science, University of Warwick

The main aim of this project is to design and develop the GPU-based algorithm to find the optimal co-scheduling solutions for multicore computers. This project is based on the graph-based method for finding the optimal co-scheduling solution for serial and parallel jobs. It is possible to parallelise the proposed algorithm in order to speed up the scheduling process using CUDA. A number of different approaches were developed. The useful features from each of them have been used to accelerate the scheduling process. The queue approach based on modified Dijkstra's algorithm showed the best result.

I Introduction

Nowadays, single-core processors rapidly reach the physical limits of possible complexity and speed, and multicore processing is becoming more widely used. Most current systems are multicore. Multicore processor refers to a single integrated circuit chip and consist of more than one processing unit. Each core or processor shares the same interconnection to the rest of the system. When multiple applications run on a multicore CPU (Central Processing Unit), they compete for shared resources, such as on-chip cache, and can suffer performance degradation. This performance degradation is called co-run degradation. In order to effectively reduce the resource contention, the content-aware co-schedulers are used. A number of co-schedulers have been developed in the literature (for example [1-4]). However, most of them do not aim to find optimal performance. Furthermore, most of them model the problem for serial jobs, where each job occupies a single core, but modern multicore systems run a mix of serial and parallel jobs simultaneously. Therefore, the ability to find an optimal solution is critical, even if it has to be obtained offline. This knowledge can help achieve a better hardware utilisation by reducing the gap between current and optimal performance of a system.

The work in [5] developed a graph-based method to solve the optimal co-scheduling problem for both serial and parallel jobs. Then the problem is modelled as finding the shortest valid path in that graph. A number of optimisation strategies were developed to accelerate the solving process. It is possible to parallelise the proposed co-scheduling algorithm to speed up the scheduling process using CUDA(Compute Unified Device Architecture), which is a programming model designed for GPUs (Graphics Processor Unit). GPUs offer a finer level of data parallelism than a CPU, with which large speedups are possible.

General purpose parallel programming on GPUs is a relatively recent phenomenon. The GPUs were originally designed to rapidly manipulate and alter memory to accelerate the creation of images and are widely used for 3D game rendering. Those capabilities are being more broadly used in various areas to accelerate computational workloads. The design philosophies of a CPU and a GPU are completely different (Fig. 1). Architecturally, the CPU has a latency-oriented design which minimises the execution latency of a single thread. It is composed of only a few cores with large last-level on-chip caches, which capture frequently accessed data and convert some of the long-latency memory accesses into short-latency cache accesses. Complicated arithmetic units (ALUs) and control logic of CPUs are designed to minimise the latency of the operation. Applications with one or very few threads achieve higher performance in the CPU. In contrast, a GPU has a throughput-oriented design which maximises the total throughput of a large number of threads while allowing individual threads to take a longer time to execute. A GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. It uses simple control logic and simple ALUs, allowing memory access and arithmetic operations to have long latency.

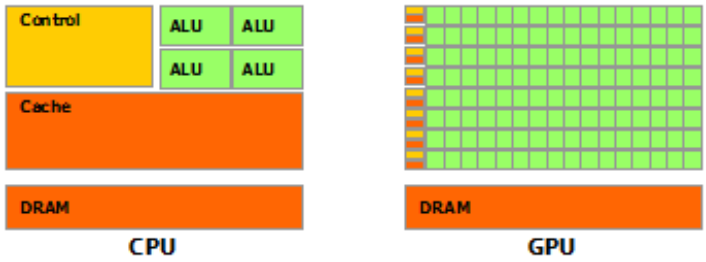

Fig. 1. Architectures of CPU and GPU (by CUDA C Programming Guide).

The architecture differences between the CPU and the GPU lead to a discrepancy in floating-point capability and memory bandwidth. Therefore the GPU is particularly suitable for problems that can be represented as data-parallel computations. A data-parallel programming model focuses on performing the same operations on many data elements in parallel with a high ratio of arithmetic operations to memory operations. In data-parallel processing, data elements are mapped to parallel processing threads. This programming model can be used to speed up the computations in applications which process large data.

The CUDA parallel programming model is based on kernel functions, which define the operation of individual threads within a thread block and a grid (Fig. 2). The last GPU chip was designed in such a way that it supports dynamic parallelism and recursion, which makes it superior to old ones.

Dynamic parallelism has been achieved by extending the CUDA programming model and enabling a CUDA kernel to create and synchronise new nested work, i.e. threads can launch more threads (Fig. 3). An application can launch a coarse-grained kernel which in turn launches finer-grained kernels to do work where needed. In this way all interesting details will be captured, while unwanted computations will be avoided. Dynamic parallelism is especially well-suited to address problems where nested parallelism cannot be avoided. For example, in cases where work is naturally split into independent batches, each batch involves complex parallel processing but cannot fully use a single GPU [6].

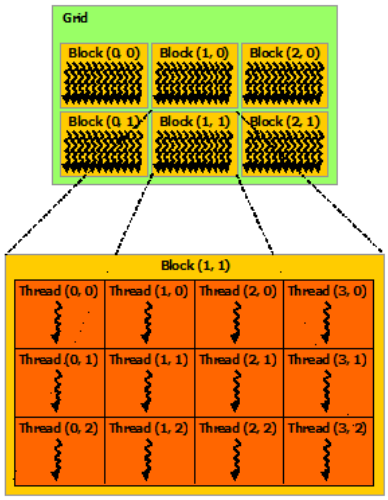

Fig. 2. Grid of thread blocks (by CUDA C Programming Guide).

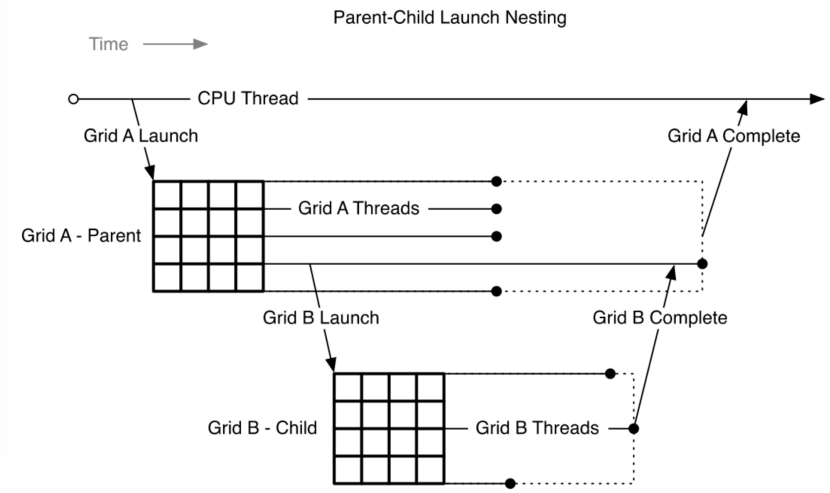

Fig. 3. Dynamic parallelism in CUDA (by CUDA C documentation).

II Methods

Formalising the job co-scheduling problem. The objective of the co-scheduling problem is to find the optimal way to partition *n* jobs into *m* *u*-cores machines, so that the sum of degradations d_i over all *n* jobs is minimised. A number of *u*-core processors *m* can be calculated as *n*/*u*, if *n* cannot be divided by *u* evenly, (*u* - *n* mod *u*) number of imaginary jobs can be added which have no performance degradation with any other jobs. To model the co-scheduling problem, the co-scheduling graph was constructed [5]. The graph consists of $\binom{n}{u}$ nodes and each node represents a *u*-core processor with *u* jobs assigned to it (which is called the node jobset). Jobs in nodes are presented in ascending order. The weight of each node d_i represents the total performance degradation of the set of *u*-jobs in that node. The graph has a number of levels from 1 to *n*-*u*+1. Each node is assigned to one particular level depending on the first job present in the node. The *i*-th level contains all nodes in which the first job is *i*. The nodes at each level are placed in ascending order based on the jobset they contain. For processing purposes, there are two extra nodes: *start* at level 0 and *end* at the last level, both with zero weights. The edges between the nodes are dynamically established as the algorithm progresses. Such organisation of the graph nodes is used for reducing the time complexity of the co-scheduling algorithm. Figure 4 illustrates the case where 6 jobs are co-scheduled to 2-cores processors: the list of numbers in each node is node jobset, the number next to the node is the node weight.

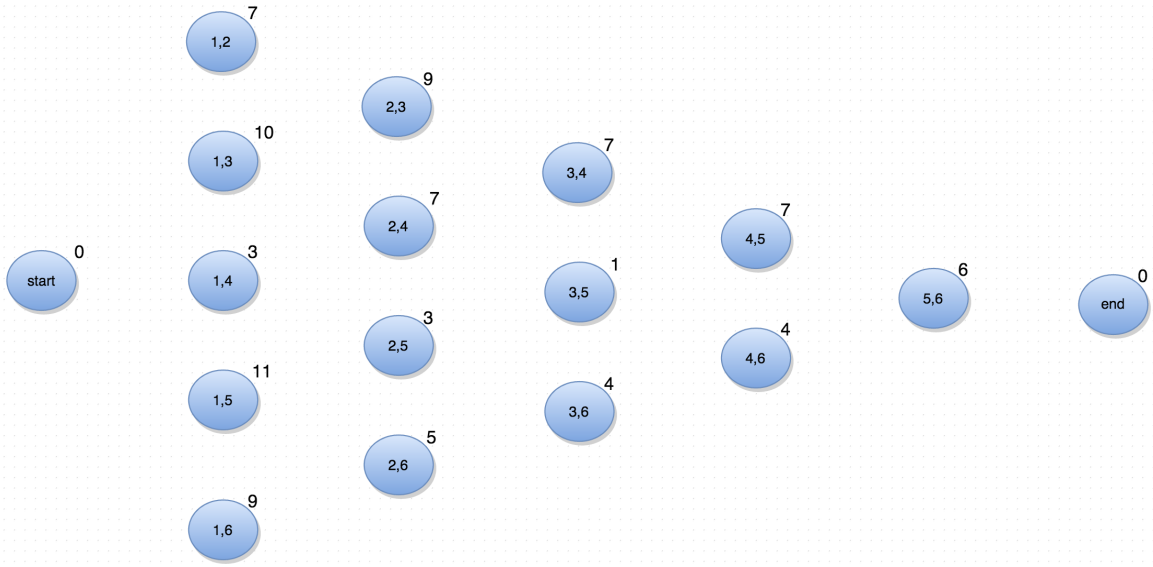

Fig. 4. Co-scheduling graph.

A valid path in the graph is a path from the *start* node to the *end* node in the constructed graph, which forms a co-scheduling solution and does not contain duplicated jobs. This means that the links can be established only forward, i.e. node at *i*-th level can be connected only to nodes at the next levels (*i*+1 and further), and not with nodes from previous levels (*i*-1 and earlier) and current level (*i*). The distance of a path is defined as the sum of the weights of all nodes on the path. Finding the

optimal co-scheduling solution is equivalent to finding the shortest valid path from the *start* node to the *end* node in the co-scheduling graph.

The most common algorithm for finding shortest paths in graphs is Dijkstra's algorithm. However, it cannot be directly applied to find the optimal solution. It can be illustrated by following example. Let us consider only a few nodes from the graph (Fig. 5). There are two valid paths, red (1,6) - (2,3) - (4,5) and green (1,5) - (2,3) - (4,6). Up to node (2,3) green path has a greater distance (20), than the red path (18), and therefore will not be examined after this node according to Dijkstra's algorithm. In order to reach *end* node, a red path has to connect to node (4,5) to form a final valid path with the distance of 25, whereas the complete green path has shorter distance, but is dismissed by Dijkstra's algorithm during the search for the shortest path.

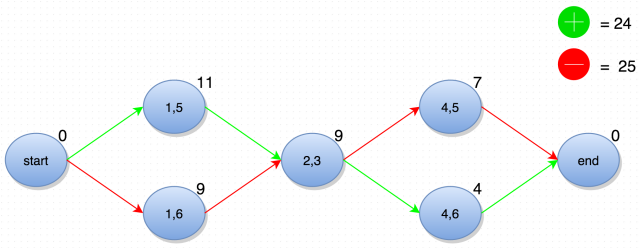

Fig. 5. Problem with Dijkstra's algorithm.

The main reason for this is because Dijkstra's algorithm dismisses some optional subpaths. After the algorithm runs up to a certain node in the graph by recording only the shortest subpaths up to it, not all nodes among the unsearched ones can form a valid schedule with recorded shortest subpath. Because of this limitation, the algorithm may connect the recorded subpath to the nodes with bigger weights, while other subpaths dismissed by Dijkstra's algorithm could be able to connect to the nodes with smaller weights and form a shorter and still valid final path.

Skipping strategy. We developed several different approaches for accelerating co-scheduling process. Because of the fact that not all paths in the graph are valid, the skipping strategy was introduced to reduce execution time of algorithms (Fig. 6). Firstly, any path which has a length less than m nodes between *start* and *end* nodes will be skipped due to the reason that it needs exactly m nodes to perform correct solution. Secondly, the current path will be skipped if it reaches an invalid node, i.e. a node which contains a job that has already appeared in the path. And finally, any path is invalid if its distance exceeds some limit. This limit was introduced as the highest expected total degradation or distance (this idea is similar to the $A*$ algorithm [7]). It can be calculated as

$$D = \alpha \times m \times \frac{1}{n} \times \sum_{i=1}^{\binom{n}{u}} d_i, \tag{1}$$

where α is a coefficient usually greater than 0 and less than 1. In the Results section it will be shown that certain choice of α value allows the algorithm to skip a lot of paths and reduce the execution time.

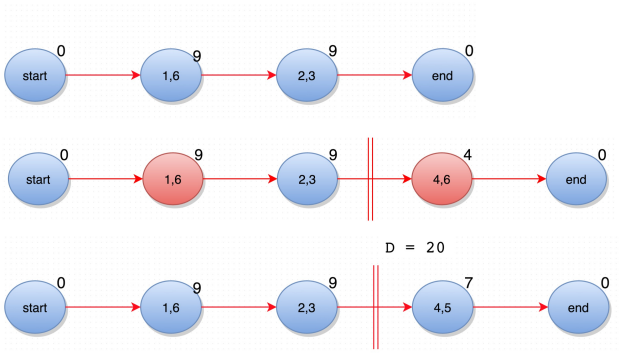

Fig. 6. Skipping strategy.

Approach A: Recursion. This recursive algorithm works based on a pre-set of nodes connected together. It uses a depth-first search to find all paths between *start* and *end* nodes. Beginning from the *start* node, the algorithm examines all outgoing links and progresses by expanding the first child node of the search tree that appears, searching progressively deeper until the *end* node is found. The search then backtracks, returning to the most recent node it has not finished exploring. The advantage of this approach is that it returns all paths between *start* and *end* nodes, which satisfy the skipping strategy. It means that if there are more than one path with minimum total degradation, this algorithm can return all of them. However, it has a few disadvantages. For this approach the initial graph should be represented in some data structure. The most common representation of a graph is adjacency matrix. The downside of such representation is its waste of space, since this matrix is sparse. Instead, an adjacency list provides more compact storage, where each vertex in a graph is associated with an array of adjacent vertices. Another drawback of this approach is recursion itself. Each iteration requires space on the call stack and will consume memory for variables and such until it completes, which will not happen until the final iteration has completed and the recursion starts to unwind. This is very likely to lead to stack or memory segment overflows which will crash an application. In addition, recursive functions are typically slower than their iterative counterpart. Due to the restrictions of CUDA, most importantly was a lack of ability to resize the used memory inside the kernel function, this approach was implemented only for running on the CPU for the purpose of validating the results of other approaches.

Approach B: Combinations. This algorithm is based on brute force search and is equivalent to enumerating all possible subpaths starting from level 1 in the graph. There are $\binom{n-1}{u-1}$ nodes on graph first level. Therefore the total number of paths starting from level 1 can be computed as

$$L = \sum_{i=1}^{\binom{n-1}{u-1}} \binom{\binom{n}{u} - i}{m-1}. \tag{2}$$

Then each path can be validated independently of others, which allows us to achieve a good level of parallelism (Fig. 7). At the last step, the algorithm finds among all valid paths the one with

minimum degradation. This approach has a number of drawbacks. Foremost is that the ratio of the number of invalid paths in graph to number of valid paths is high, which makes this approach very inefficient. But the idea found in this approach of reconstructing a path (and node jobset) from its index number in a list of all possible combinations allowed us to improve the performance of other algorithms.

#	path	validation	distance
1	(1,2) - (1,3) - (1,4)	invalid	-
2	(1,2) - (1,3) - (1,5)	invalid	-
3	(1,2) - (1,3) - (1,6)	invalid	-
...	...		
83	(1,2) - (3,5) - (4,6)	valid	12
84	(1,2) - (3,5) - (5,6)	invalid	-
85	(1,2) - (3,6) - (4,5)	valid	18
...	...		
210	(1,4) - (2,5) - (3,6)	valid	10
...	...		
334	(1,6) - (4,5) - (5,6)	invalid	-
335	(1,6) - (4,6) - (5,6)	invalid	-

Fig. 7. Combinatorial approach.

Approach C: Queue. This algorithm is based on Dijkstra's algorithm with modifications. Instead of using a vanilla version of the algorithm, which cannot be directly applied to find the optimal solution, it needs to revise Dijkstra's algorithm so that it will not dismiss any subpath, i.e. to record every subpath visited by the algorithm. Implementation of this approach for the CPU is exactly the same as in paper [5]. However, it cannot be directly parallelised and therefore the algorithm was modified.

At the first step all nodes from first level are in the array Q `(Fig. 8)`. This structure is an input for loop, which has a m-1 iterations, each of them will expend subpath with one node. The next several steps are processed for each entry of Q: find a next valid level, identify a set of valid nodes. After that a temporary array T is filled using the rule: add valid node along with its parent subpath from Q. If this node performs a new subpath which contains the same jobset as one of the entries of T and has smaller weight, then replace the old subpath with new, otherwise skip the new subpath. At the next step the T and the Q are swapped. This Q is an input for the next loop iteration and contains a subpaths one node longer. When the loop is finished, it is the only one path with the smallest degradation in T. The idea behind this algorithm is the same as in the sequential version, but it allows us to perform calculations inside loop in parallel for each entry of Q.

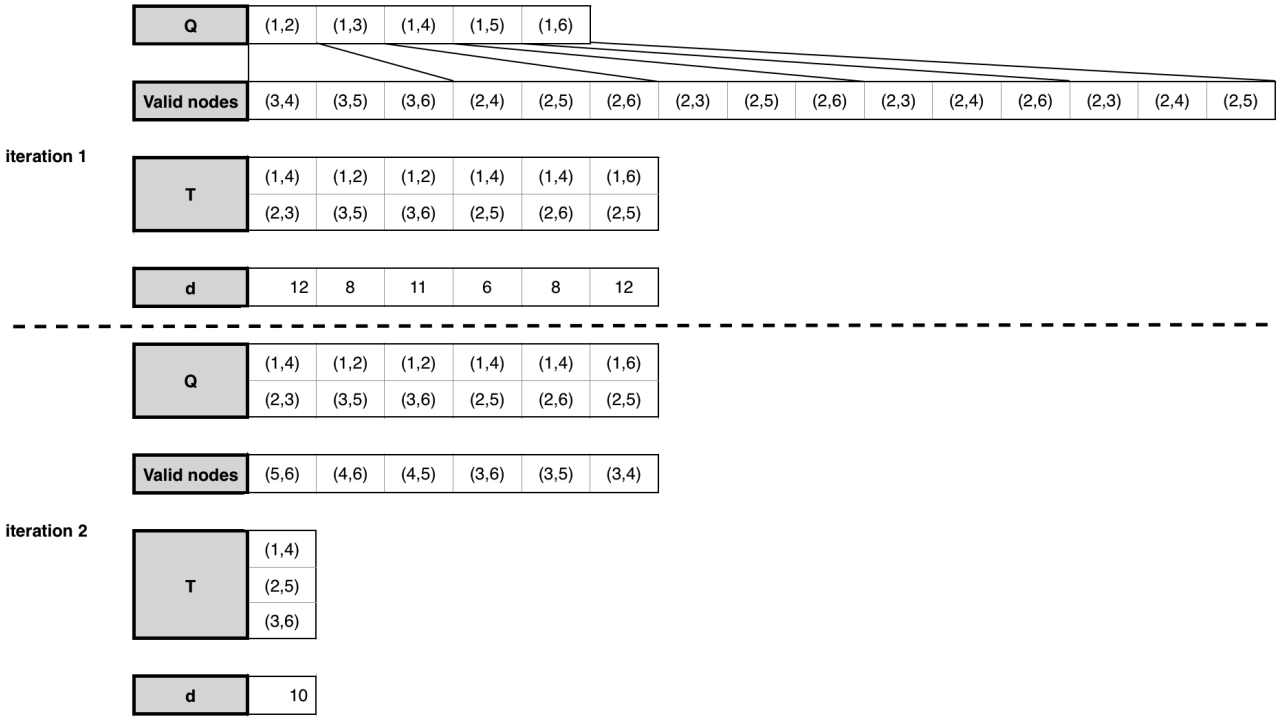

Fig. 8. Approach with a queue.

III Results

As described above, the highest expected total degradation or distance can reduce the number of valid paths. The value of coefficient α in (1) controls the skipping rate. This value should be small enough to allow the algorithm to skip as many paths as possible, but it should be reasonably big, otherwise the algorithm will not be able to find any valid path. Figure 9 shows a minimum value of α for different graph. Setting the right value of α leads to skipping of more than 80% of potentially valid paths, which is speeding up the scheduling process.

Figure 10 presents a time comparison for two approaches: queue (modified Dijkstra's algorithm) and combinations (brute force search). Brute force search can be efficient for small graphs with a small number of cores, because the number of paths in the graph are reasonably small. When graph size increases, the total number of paths grows rapidly (according to (2)), therefore execution time grows. On the contrary, a queue can produce a result relatively faster. It should be noted that figure 10 shows the result of the earliest version with a queue which does not contain a number of optimisation strategies developed later to accelerate this algorithm.

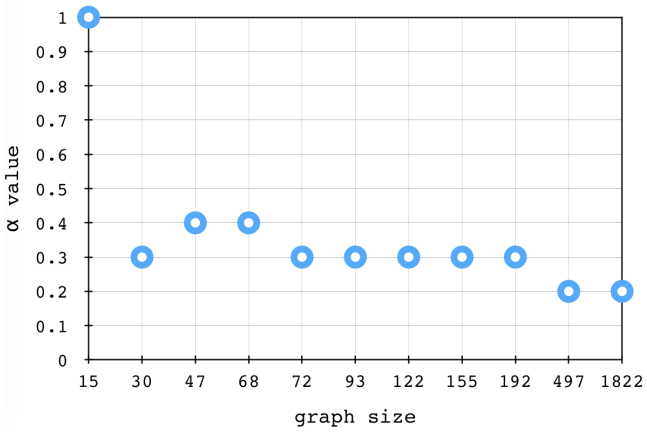

Fig. 9. Effect of α value.

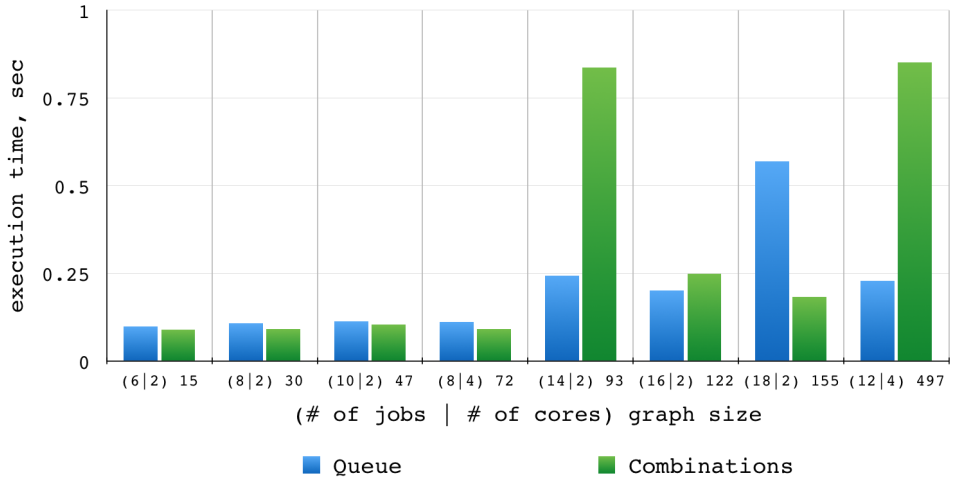

Fig. 10. Time comparison.

IV Discussion

A number of different approaches were developed for the GPU-based algorithm to find the optimal co-scheduling solutions for graph-based method. Each of them gave some useful ideas for accelerating the scheduling process. The only downside is that the performance is not as good as expected. It indeed needs some fine-tuning to achieve better performance. At this point the queue approach shows better results, perhaps because it is based on the fastest sequential solution - Dijkstra's algorithm. It is possible to speed up the scheduling process further using following ideas. The number of copies of the memory being copied to/from the kernel needs to be reduced because it slows down the process. A static memory allocation (memory is assigned during compilation time) is more efficient in terms of speedup and should be used instead of a dynamic memory allocation (memory is assigned during run time). The nodes skipping strategy [5] should be taken into consideration. Memory usage can

be reduced by using the idea from the combinatorial approach, where the jobset of any node can be reproduced by its index within the graph. The benefits of CUDA dynamic parallelism can help to reduce execution time.

V Acknowledgments

I would like to express my thanks to my supervisor Dr. Ligang He and PhD student of Computer Science department University of Warwick Huanzhou Zhu for their guidance and help provided during the project. Furthermore, I am grateful to the Erasmus Mundus consortium and everyone who takes part in organising the Programme in Complex Systems Science.

VI Reference

1. S. Zhuravlev, S. Saez, J. Cand Blagodurov, A. Fedorova, and M. Prieto, *Survey of scheduling techniques for addressing shared resources in multicore processors,* (ACM Computing Surveys (CSUR), 2012).

2. S.Blagodurov, S.Zhuravlev, A.Fedorova, and M.Dashti, *A case for numa-aware contention management on multicore systems,* (in In Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, 2010).

3. P. Apparao, R. Iyer, and D. Newell, *Towards modeling & analysis of consolidated cmp servers,* (SIGARCH Comput. Archit. News, 2008).

4. Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi, *The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions,* (Parallel and Distributed Systems, IEEE Transactions on, 2011).

5. H. Zhu, L. He and S. A. Jarvis, *Optimizing Job Scheduling on Multi-core computers,* (in IEEE 22nd International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems).

6. A. Adinetz, *Adaptive Parallel Computation with CUDA Dynamic Parallelism,* (http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/).

7. P. E. Hart, N. J. Nilsson, B. Raphael, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths,* (Systems Science and Cybernetics, IEEE Transactions on, 1968).