An Approach to Parallelizing the Simulation of Complicated Modelica Models

Joshua D. Carl, Gautam Biswas, Sandeep Neema, and Ted Bapty
Institute for Software Integrated Systems, Vanderbilt University
Nashville, TN 37235
{carljd1, biswas, sandeep, bapty}@isis.vanderbilt.edu

Keywords: equation-based modeling, directed acyclic graph, parallel scheduling, efficient simulation models

Abstract

Designing embedded systems has become a complex and expensive task, and simulation and other analysis tools are taking on a bigger role in the overall design process. In an effort to speed up the design process, we present an algorithm for reducing the simulation time of large, complex models by creating a parallel schedule from a flattened set of equations that collectively capture the system behavior. The developed approach is applied to a multi-core desktop processor to determine the estimated speedup in a set of subsystem models.

ACKNOWLEDGMENTS

This work was partially funded by DARPA contract HR0011-13-C-0041.

1. INTRODUCTION

The complexity and cost of the design process for systems, such as vehicles, aircraft, and power plants, makes modeling and simulation tools a critical component of the modern design process, and it is clear that their importance will continue to grow in the future. When applied in an effective manner, simulation can be used to decrease the overall design time; verify design specifications and constraints; identify, plan and leverage emergent behavior across design components and domains; and ease the addition of requirements and new technology to the design [13] [15]. By integrating simulation into the design process we are looking to significantly reduce the time and cost to design a new product.

Our simulation task requires translating the declarative, physics-based, model that the designer creates into a simulation model, which, in our work is a partially ordered set of equations. These ordered equations may be represented as a directed acyclic graph (DAG), called a task graph, with each node representing one or more equations. The goal of this paper is to use the task graph to create blocks of equations and to create parallel and sequential relations among these blocks. The task of generating an execution schedule for the equation blocks has been studied in previous work (such as: [12], [8], [3], and [14]) and we plan to use these algorithms to create our parallel task schedule, and therefore decrease the simulation time of a complex model. In this paper we will look at one approach to parallelizing the simulation of a complex system model to determine the amount of speedup we can expect compared to execution on a single processor.

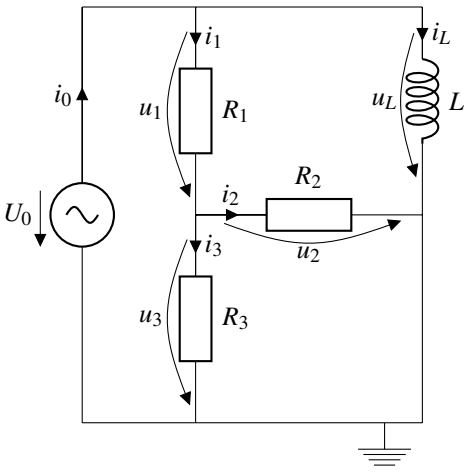

Figure 1. Circuit example

Recently, there has been a rise in the parallel processing power available to a system designer. Modern engineering workstations typically have multi-core processors, and cloud computing options have become more widespread. We are interested in leveraging this parallel processing power to reduce the computation time for each simulation run. This will result in a more efficient design-verify feedback loop for the system design engineer.

The paper is organized as follows: Section 2. provides some necessary background to modeling physical systems and parallel algorithms. Section 3. describes algorithm we will use to parallelize the simulation. Section 4. provides a case study of our methods. Section 5. presents our conclusions and future work.

2. BACKGROUND

This section reviews the component-oriented modeling tools that form the framework for generating the flat equation-based model of the system. We also review the literature on parallel scheduling algorithms that have been developed for scheduling a set of partially ordered equations describing system models.

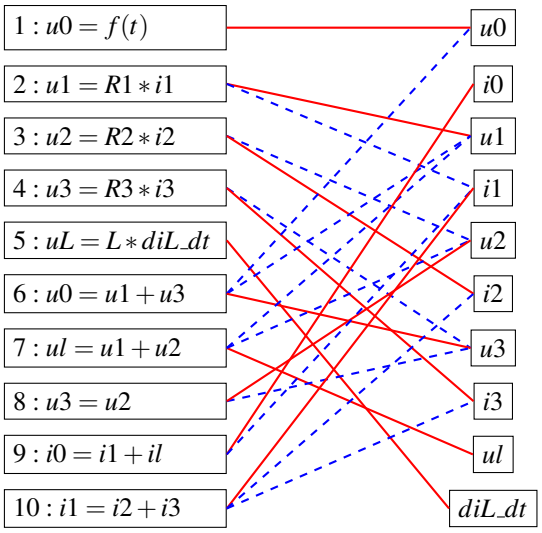

Figure 2. Causal equations

2.1. Modeling Tools

We use the Modelica language [1], an equation based programming language designed to model physical systems, to simulate system behavior. It has traditional programming language constructs such as functions, parameters, inheritance, logical control statements (if-else), and loops. Models exchange data through connectors, and each connector defines the terminal variables. Submodels can be added to a larger model with the interactions happening through the connectors. Since Modelica is equation based, the behavior of the components and the interactions between components are acausal. The causality of the system equations is determined by the model compiler, and not by the model designer. This is important for component based design, as it means that component models do not need to be created for every causal situation, simplifying the model construction task.

2.2. Transforming Equations to a Task Graph

All Modelica compilers perform roughly the same high-level tasks to transform a hierarchical declarative model to an executable model, where the executable model is a partially ordered graph of computational blocks. As a running example, we will reference the circuit in figure 1 (this example circuit and equations are taken from [7]). First, all hierarchy is removed from the model so that the model is reduced to a completely flattened set of equations. Second, a bipartite graph of equations and variables is created. There are links between the equations and variables if a variable is used in an equation. Note that any state variables should be treated as constants in the graph (no link to an equation) and the state variable derivative is treated as a variable. Third, perform a matching between the equations and variables using a maximum flow algorithm [11] [9]. This causalizes the system and determines which equation solves for each variable. There is a one-to-one mapping of equations and variables, so that each equation solves for only one variable. Note that if the system is not at index 1, then perform index reduction using Pantelides algorithm [20] (a description is included in [7]). The bipartite graph for the circuit in figure 1 is shown in figure 2. The solid red lines are the causal equation/variable pairs and the blue dashed lines are the equations that use the variable, but do not solve for it. This system is of index 1, so there is no need to perform index reduction. Fourth, create a directed graph from the bipartite graph by converting each non-matching edge into a directed edge pointing from the variable to the equation it which it is used. Then collapse each equation-variable matching pair into a single node. This is shown in figure 3. Fifth, run Tarjan's algorithm [22] to determine the strongly connected components (SCCs). These represent the systems of equations. Finally, collapse each group of equations into a single strongly connected component node. The graph in figure 3 has a strongly connected component with 6 variables tied up in the loop; equations 6, 8, 4, 3, 10, and 2. Finally, collapse the strongly connected component into a single node. The final DAG is in figure 3.

At the termination of the final step, the hierarchical declarative model has been transformed into a directed acyclic graph. Each node in the graph represents one SCC, and each SCC represents the computational work of one or more equations. If the number of equations in the SCC is more than 1, then it means that the SCC represents an algebraic loop. The hierarchy in the graph represents the order in which the equations need to be solved, to guarantee that each equation is using the most current information when calculating the value of a variable.

This approach does not include the integration algorithm as a part of the system of equations, which is known as inline integration, and has been studied before in [7] and elsewhere. Inlining an explicit integration algorithm in a parallel environment will likely produce an extra speedup that is not seen when the algorithm does not use inlining because it merges two formerly separate processes: calculating the system derivatives, and integrating those derivatives. Inlining an implicit algorithm may not improve the simulation time directly, as implicit algorithms add a non-linear algebraic loop to the system equations [7]. However, inlining an implicit algorithm should not be written off, because inline integration of an implicit algorithm may allow for direct DAE simulation, which avoids the problem of needing to perform index reduction during the model compilation process. There are numerous other advantages to directly simulating the DAE equations, and [7] is recommended for further reading.

2.3. Parallel computation

As described in the previous section, the end result of a Modelica model compilation process is a a directed acyclic

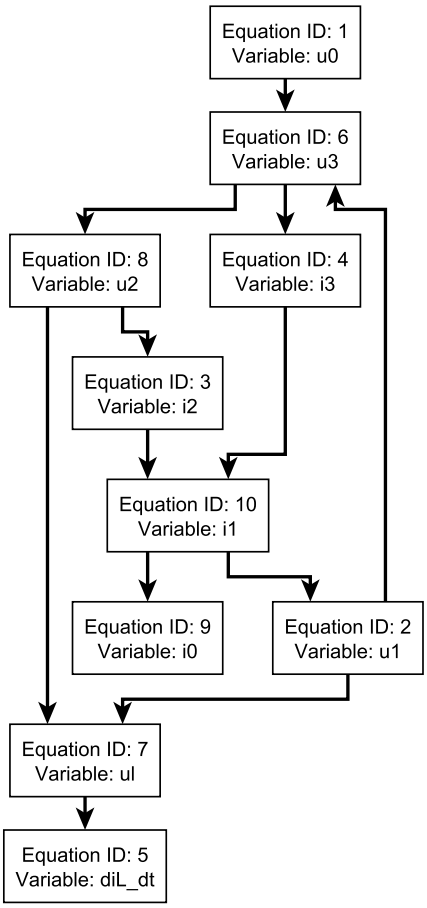

Figure 3. Directed graph of equations from figure 1

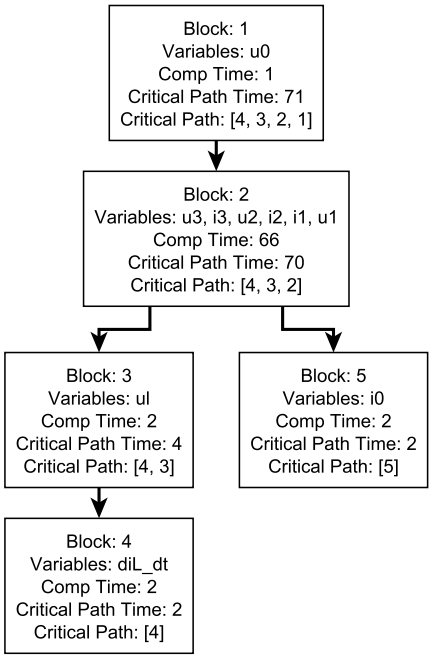

Figure 4. Directed acyclic graph of equations from figure 1

task graph, where each task in the graph represents one or more equations to be solved. Parallelizing this task graph to speed up the simulation is our primary interest.

Outside of certain specific situations, creating an optimal schedule in this scenario has been shown to be NP-complete [14], but there are a number of heuristic algorithms have been developed that are known to generate good schedules that speed up computation. A good schedule is very important because it guarantees that the tasks will be completed in order, that the processor workload is balanced with minimal amounts of processor idle time, and that there is a speed-up compared to a single processor environment.

2.3.1. Scheduling Algorithms

Hu [12] presented an optimal algorithm for assigning tasks to an unlimited number of processors. The drawbacks to his approach are that each task was assumed to take the same time, and the graph representing the system had to have an in-tree structure, where each node could have multiple predecessors but only one successor. The algorithm works by always trying to schedule the tasks that are the farthest away from the terminal node.

Coffman and Graham [8] presented an optimal algorithm for scheduling nodes on an arbitrary tree, but only on a system with 2 processors, and, again, with tasks that are all the same length. The nodes are scheduled based on their distance from the terminal node, and if there is ever a tie between two nodes, then the algorithm will pick the node where its set of successors is a superset of the other node. If the sets of successors are disjoint, then a node can be chosen arbitrarily.

Adam, Chandy, and Dickson [3] presented a comparison of various list scheduling algorithms that supported task graphs of any arbitrary size and shape (as long as they were directed and acyclic), and where the tasks in the graph may have any computation time. A list scheduling algorithm creates a list of the tasks sorted by priority. As tasks become available to run, the task with the highest priority is based on the priority list. In their experiments the highest level first with estimated times (HLFET) algorithm, where the tasks with the highest level are given scheduling priority, consistently produced the best results. Their highest level algorithm was implemented as the critical path (CP) algorithm in other papers [14], and is defined as the longest path from the current node to the terminal node taking into account the task processing time and the task communication time. This represents a lower bound on the length of time to solve the system. In [3] the CP algorithm created a schedule that was within 4.4% of the optimal schedule in 265 out of 266 test cases. Kasahara and Narita [14] presented a refinement to the HLFET algorithm by including a tie breaker condition so that if there every was a tie between two tasks the task with the most immediate successors is scheduled first (CP/MISF).

Of these algorithms the CP and CP/MISF are the most relevant to our purposes. They produce good schedules, and they do not have the restrictions of Hu's algorithm and Coffman and Graham's algorithm.

2.3.2. Modelica Parallelization

OpenModelica (OM) [2] is an open source Modelica compiler. The team behind OM have done a fair amount of research for parallelizing Modelica models by dividing the processing across multiple networked PCs. Some early work was presented in [4]. They proposed breaking each of the model equations down into the individual arithmetic operations and treating each operation as a task in the task graph. Then the different tasks are merged or duplicated to produce a task graph that is easily parallelizable. However, the integration algorithm was not parallelized, so once all of the equations were solved for a time step, the solved variables were sent back to the main CPU running the solver to perform the integration [16]. They also presented some later work with the same equation parallelization scheme, but also included a parallel solver [16]. In [17] they presented an algorithm to sort the tasks assigned to the various processors to minimize communication delays.

They have also explored using a GPU to perform the simulation [19] and have investigated using transmission line modeling [18] [21] to add physically accurate delays to the system. These delays allow the system to be broken into independent component at the delay points.

More recently, Casella [6], proposed an algorithm specifically for creating a parallel schedule for a Modelica model. The algorithm starts by grouping the tasks that have no predecessors into a set S_1. These tasks are removed from the task graph, and the new nodes that have no predecessors are grouped into a set S_2. The process continues until all of the nodes in the graph are assigned to one of the sets of S_i. The sets are scheduled in order, with the tasks in each set scheduled in a random order. The simulation must wait until all of the tasks in each set are complete before beginning to process the next set. This slows down the overall processing as some tasks from a later set will likely be able to start before all of the tasks from the current set finish.

In [10] Elmqvist, Mattsson and Olsson presented a refinement of Casella approach to creating a parallel schedule. The authors used heuristic rules to divide the system of equations into several layers. Within each layer a number of sections are created that may be executed in parallel. Tasks are assigned to a section that must be executed in sequence. They achieved speed up factors of 2.1 to 3.4 on a variety of systems, where the speedup factor is defined as the serial execution time divided by the parallel execution time.

Walther, et. al. [23] also presented an approach to parallelizing a Modelica model based on the dependency graph of the system. Their scheduling approach required them to estimate the communication and execution times for the tasks. The communication costs were based on measured communications between processors, and then applied to the system based on how much data needed to be transported between tasks. The execution times for each task were measured by running the model serially. The measured task times were then applied to the dependency graph for scheduling. In order to help reduce the communication costs between the tasks, they merge tasks in certain situations. In their results the best cases were almost 5 times faster than serial processing.

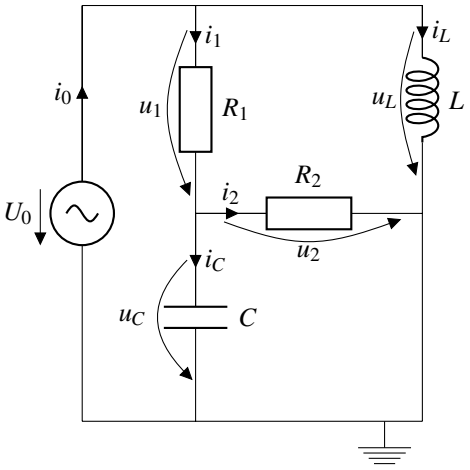

Figure 5. Second circuit example

We are targeting a modern engineering workstation with a multiple core processor as our simulation environment. This will allow us to have a shared memory across all processors, and to treat communication times as constant or negligible. This parallel architecture means there is little to be gained from breaking equations into pieces and restructuring the graph to reduce the communication costs between the tasks. Casella's approach is similar to ours, but we are focused on applying a traditional scheduling algorithm to a task graph derived from a Modelica model.

3. APPROACH

In this section we will define task graph creation algorithm, describe our task computation cost estimation process, and present a simple example.

3.1. Algorithm and Example

We are now ready to present an algorithm to populate the task graph with the necessary information to build a schedule. In parallel to describing the algorithm we will look at the two simple electrical circuits (both examples are taken from [7]). The task graph for the first circuit is shown in figure 4 and was derived in the previous section. The second circuit is shown in figure 5. The task graph for the second circuit is in figure 6. Both examples have 10 equations, except that R_3

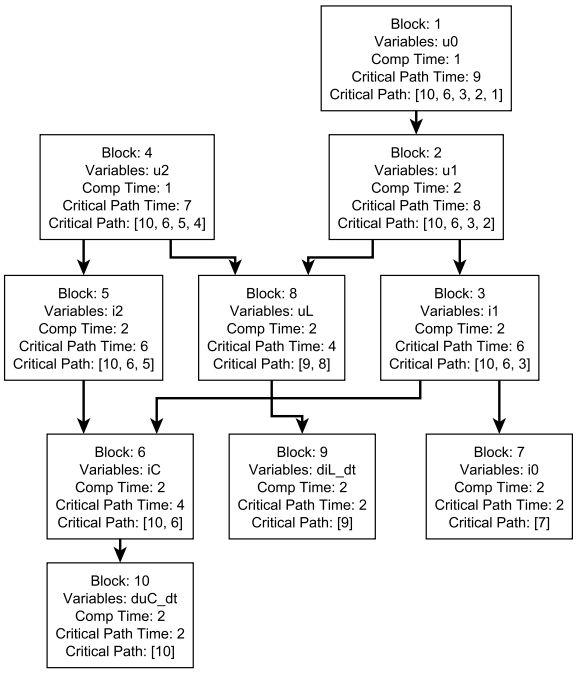

Figure 6. Directed acyclic graph of equations from figure 5

in the first example has been exchanged for a capacitor in the second example. This means that the circuit in figure 1 has an algebraic loop, while the circuit in figure 5 does not. This will have a significant impact on the estimated computation times of each node in the graph.

This algorithm estimates the computation times and calculates the critical path value of each node. First, create a copy of the graph, initialize the critical path value in all nodes to 0, and create a global empty task_priority_list. Second, identify all of the leaf nodes in the copy, nodes without any successors, in each DAG. In figure 4 the leaf nodes are blocks 4 and 5, and in figure 6 the leaf nodes are blocks 7, 9 and 10. Third, estimate the computation time for each leaf node, and add the estimated time to that node's CP in the original graph and add the node information and CP to the appropriate place in the task_priority_list. In figure 4 blocks 4 and 5 have an estimated computation time of 2. The leaf nodes in figure 6 also have an estimated computation time of 2. Fourth, take the calculated CP and compare it to the CP of each of the node's parents. If the child's CP is larger than the parent's CP, store the child's CP as the parent's CP in the original graph. Fifth, remove the leaf nodes and their connections from the copy graph. Finally, repeat the algorithm from step 2 until all of the nodes have an estimated CP, and there are no more nodes in the copy graph. By the end of the algorithm we have a directed acyclic graph, where each node in the graph has an estimation of its critical path value and a knowledge of its predecessor and successor nodes. The scheduling algorithm is now free to begin scheduling the tasks on the available processing nodes based on the critical path value of the tasks in the task_priority_list.

The block representing an algebraic loop, block 2, has an estimated time of 66, and is almost 10 times longer than the other 4 tasks combined (combined estimated computation time of 7). This means that block 2 will dominate any parallel scheduling algorithm. In a two processor system, block 1 and block 2 will be scheduled sequentially on one processor. The other processor must wait idle for 67 time units until block 2 is complete. When block 2 is finished, blocks 3 and 5 may be scheduled in parallel, and block 4 may be scheduled on either processor when block 3 is complete. This gives a total schedule time of 71. Due to the simplicity of the graph and the discrepancy of estimated computation times due to the algebraic loop, parallelization only reduces the simulation by 2 time units compared to the single processor time (single processor time of 73).

The second example has no algebraic loops, and all of the tasks have estimated processing times of 1 or 2. This eases the task of creating a schedule with few gaps. On 2 processors, this graph produces a schedule of length 9, which is a 50% improvement over a single processor schedule, which has a schedule of length 18. The 2 processors parallel schedule is an ideal schedule since there are no idle time periods.

3.2. Task Computation Cost Estimation

The scheduling algorithms we used require an estimation of the task processing time. We use a relatively simple scheme of estimating the task computation time. Each simple arithmetic operation (addition, subtraction, multiplication, division, and logical comparisons) is assigned a value of 1. If this is a simple equation (not a system of equations) then the total number of arithmetic operations is the computation cost. For systems of equations, the estimating the computation time is more complex because systems of equations take longer to solve than simple equations. Linear systems of equations can be solved relatively efficiently, but require computational overhead above a simple set of equations. Non-linear systems of equations may require a number of iterations through the system of equations, and will therefore take longer to solve than a linear system of equations. Our task estimation reflects this increased time by multiplying the total number of arithmetic operations in the system of equations by the number of equations if it is a linear system of equations, and by the square of the number of equations if it is a non-linear system of equations.

4. CASE STUDY

To study the relative speed up gained by processing the model equations in parallel, we will look at two case studies.

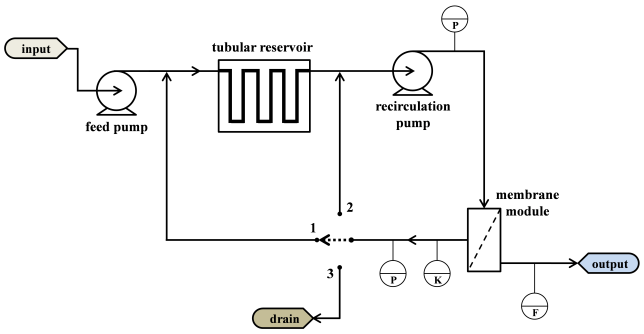

Figure 7. RO Subsystem Schematic

Statistic	Value
Equations	28
Variables	28
Blocks	26
Non-linear blocks	0

Table 1. Key statistics for the RO system.

4.1. Reverse Osmosis System

The RO system is part of the Advanced Water Recovery System (AWRS), which is a subsystem of the NASA Advanced Life Support System (ALS) [5]. The ALS was designed as a way to support life for extended duration space missions by reclaiming waste water. The RO subsystem uses a membrane to remove inorganic matter and particles from water (figure 7). The RO system has three different modes of operation that are controlled with a three-way valve, where each position of the valve specifies a different mode of operation. During the first two modes of operation, identified as M1 and M2, clean water leaves the system through the membrane, but dirty water, brine, is recirculated in a feedback loop to be filtered again. As a result of the feedback, the concentration of impurities in the water increases with time until all of the water must be purged from the system, during mode P, to be processed by a different subsystem. To keep our modeling task simple, we will only deal with Mode 1 of the RO system.

The RO system has 28 variables and equations, and 6 state variables. The equations of the model were derived from the bond graph presented in [5]. Information on the model size is in table 1.

It is easy to see from table 2 that parallelizing the processing of the equations decreased the length of the simulation compared to the single processor case. The schedule length was calculated by summing all of the time steps in the schedule for a single processor (in a parallel schedule each of the processors have the same length schedule), including both busy and idle periods. The percent processor idle was calculated for each processor by summing the idle time steps on each processor, and then dividing that sum by the full schedule length. It is intended to measure how efficiently the schedule uses the available processors. The speedup ratio compared to a single processor schedule was calculated by the equation

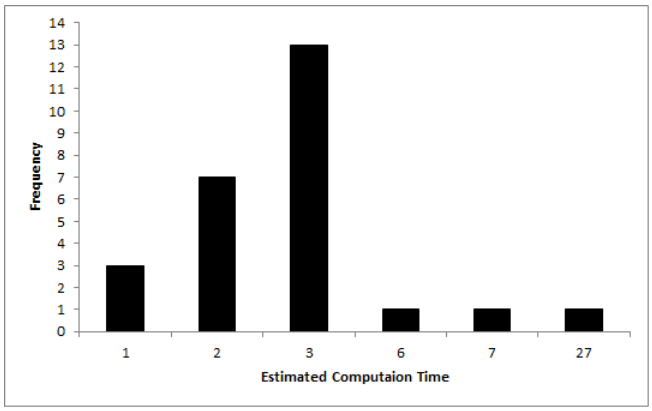

Figure 8. Frequency of task costs for the RO system

$t_{single}/t_{parallel}$, where $t_{parallel}$ is the length of the parallel processor schedule, and t_{single} is the length of the single processor schedule. The Casella algorithm is as described in [6], and CP is the critical path algorithm.

The best results were for the CP algorithm with 4 and 8 processors (both produced the same length schedule); these cases gave a 2.5 speed ratio up compared to the single processor case. However, the 4 processor case had 2 processors with and idle time for more than 50% of the schedule, and the 8 processor case had 4 processors that were idle 95% of the schedule. This is not a good use of resources. The 3 processor case seems to provide an ideal compromise between schedule improvement and ensuring high resource utilization. It provided a schedule that was almost as fast as the 4 and 8 processor case (only 1 time step difference) and there were dramatically fewer gaps in the schedule. The 2 processor case for the CP algorithm provided an ideal schedule as there are no gaps in the schedule and a 50% reduction in schedule length. The Casella algorithm did not produce as short of a schedule as the CP method. It took the Casella algorithm 4 processors to get a 50% improvement. That was accomplished with only 2 processors by the CP algorithm. Finally, to show the difference in estimated task computation time figure 8 shows how frequently a computation time was assigned. This shows that there is a wide range of task sizes of the simulation model, and that any parallel simulation structure needs to take these differences into account.

4.2. Vehicle Drivetrain

The vehicle drive train example was created by Modelon AB as a part of DARPA's AVM project. Its purpose is to provide a simple model of a vehicle's drive train. It has an engine, power take off torque converter, cross drive transmission (two "output" shafts), and drive shafts for the left and right sides of the vehicle. It is a fairly typical example of a model an engineer would create to template a vehicle drive line. The drive line equations were processed from an OpenModelica generated XML file.

Algorithm	Processors	Schedule Length	Percent Processor Idle	Speedup Ratio
Single	1	96	0	-
Casella	2	62	0.06, 0.39	1.55
Casella	3	51	0.06, 0.55, 0.51	1.88
Casella	4	49	0.14, 0.55, 0.61, 0.73	1.96
Casella	8	46	0.17, 0.74, 0.63, 0.78, 0.89, 0.78, 0.96, 0.96	2.09
CP	2	48	0, 0	2
CP	3	40	0.03, 0.25, 0.33	2.4
CP	4	39	0, 0.44, 0.51, 0.59	2.46
CP	8	39	0, 0.54, 0.54, 0.67, 0.95, 0.95, 0.95, 0.95	2.46

Table 2. Speed-up results for RO system.

Algorithm	Processors	Schedule Length	Percent Processor Idle	Speedup Ratio
Single	1	243547	0	-
Casella	2	227202	0.06, 0.87	1.07
Casella	3	226687	0.09 ,0.90, 0.94	1.07
Casella	4	226481	0.08, 0.96, 0.93, 0.96	1.08
Casella	8	226072	0.40, 0.96, 0.99, 0.69, 0.99, 0.97, 0.92, 0.99	1.08
CP	2	136596	0, 0.22	1.78
CP	3	136583	0, 0.54, 0.67	1.78
CP	4	136577	0, 0.84, 0.54, 0.84	1.78
CP	8	136577	0, 0.93, 0.55, 0.95, 0.91, 0.95, 0.99, 0.95	1.78

Table 4. Speed-up results for vehicle driveline.

Statistic	Value
Equations	816
Variables	816
Blocks	624
Non-linear blocks	126

Table 3. Key statistics for the vehicle driveline.

The vehicle drive line example provided some interesting estimated results. The results are calculated the same way as in table 2, and are shown in table 4. For the CP algorithm, increasing the number of processors beyond 2 did not significantly decrease the time it would take to process the equations, and the amount of time each processor is idle increases dramatically for each processor added. The most idle time is for the 8 processor system where 6 of the 8 processors were idle for more than 90% of the schedule. This is unexpected, but it shows that there is not much inherent parallelism in the system equations. The Casella algorithm did not perform very well as each processor configuration had a speedup ratio of just above 1. This is again likely due to the lack of parallelism in the model and to the algorithm not taking into account differences in task execution.

5. CONCLUSION AND FUTURE WORK

From the above case study it is easy to see that parallelizing the calculation of the system equations can yield a significant speed up in the total time to solve the equations. The speed up shown here will not be fully reflected in a real system because this analysis does not parallelize the integration algorithm. However, even without parallelizing the integration algorithm these methods will produce results in the form of lower simulation times.

For future work we plan to improve our cost estimation algorithm, create a parallel simulation environment, and create an on-line scheduling algorithm that will allow us to dynamically change the estimated task execution time, and therefore the critical path, during simulation into a measured task execution time. These changes will allow us to more accurately determine the benefits parallelization will have. We are also interested in fully supporting inline integration in our simulation environment. All of these techniques will reduce the time it takes to perform a simulation, which will greatly benefit an engineer that is designing and simulating a complex system.

REFERENCES

[1] Modelica. https://www.modelica.org/.

[2] Openmodelica. http://www.openmodelica.org.

[3] T. L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, Dec. 1974.

[4] P. Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Linköping University, PELAB, The Institute of Technology, 2006.

[5] J. D. Carl, Z. Lattmann, and G. Biswas. Modeling and simulation semantics for building large-scale multi-domain embedded systems. In *27th European Conference on Modelling and Simulation*, Norway, May 2013.

[6] F. Casella. A strategy for parallel simulation of declarative object-oriented models of generalized physical networks. In H. Nilsson, editor, *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, April 2013.

[7] F. E. Cellier and E. Kofman. *Continuous system simulation*. Springer, 2006.

[8] J. Coffman, E.G. and R. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[9] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, Apr. 1972.

[10] H. Elmqvist, S. E. Mattsson, and H. Olsson. Parallel model execution on many cores. In *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, March 2014. Modelica.org.

[11] D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 2010.

[12] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):pp. 841–848, 1961.

[13] G. Karsai and J. Sztipanovits. Model-integrated development of cyber-physical systems. In U. Brinkschulte, T. Givargis, and S. Russo, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 5287 of *Lecture Notes in Computer Science*, pages 46–54. Springer Berlin Heidelberg, 2008.

[14] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *Computers, IEEE Transactions on*, C-33(11):1023–1029, 1984.

[15] Z. Lattmann, A. Pop, and et. al. Verification and design exploration through meta tool integration with openmodelica. In *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, March 2014. Modelica.org.

[16] H. Lundvall and P. Fritzson. Automatic parallelization of object oriented models across method and system. In *In Proceedings of the 6th Eurosim Congress*, 2007.

[17] H. Lundvall, K. Stavåker, P. Fritzson, and C. Kessler. Automatic parallelization of simulation code for equation-based models with software pipelining and measurements on three platforms. *SIGARCH Comput. Archit. News*, 36(5):46–55, June 2009.

[18] K. Nyström and P. Fritzson. Parallel simulation with transmission lines in modelica. In *Proceedings of the 5th International Modelica Conference (Modelica'2006)*, 2006.

[19] P. Ostlund, K. Stavaker, and P. Fritzson. Parallel simulation of equation-based models on cuda-enabled gpus. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM.

[20] C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

[21] M. Sjölund, M. Gebremedhin, and P. Fritzson. Parallelizing equation-based models for simulation on multicore platforms by utilizing model structure. In A. Darte, editor, *Proceedings of the 17th Workshop on Compilers for Parallel Computing*, July 2013.

[22] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[23] M. Walther, V. Waurich, and C. S. D.-I. I. Gubsch. Equation based parallelization of modelica models. In *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, March 2014. Modelica.org.

AUTHOR BIOGRAPHIES

JOSHUA D. CARL is currently a graduate student at Vanderbilt University and is a member of the Institute for Software Integrated Systems (ISIS). His research interests include cross domain modeling and simulation of embedded and cyber physical systems. His email address is: carljd1@isis.vanderbilt.edu.

GAUTAM BISWAS is a Professor in the EECS Department and a Senior Research Scientist at ISIS at Vanderbilt University. His primary research interests are modeling and simulation of complex systems and applying these models for diagnosis and fault adaptive control. His email address is: biswas@isis.vanderbilt.edu.

SANDEEP NEEMA is a Research Associate Professor of Electrical Engineering at ISIS at Vanderbilt University. His primary research interests are embedded systems, design-space exploration, and model integrated computing. His email address is: neemask@isis.vanderbilt.edu.

TED BAPTY is a Research Associate Professor of Electrical Engineering at ISIS at Vanderbilt University. His email address is: bapty@isis.vanderbilt.edu.