Indian Journal of Science and Technology, *Vol 8(8), DOI: 10.17485/ijst/2016/v9i8/61035, February 2016*
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

A Modular Approach to Random Task Graph Generation

Mishra Ashish[1*], Sharma Aditya[1], Verma Pranet[1], Abhijit R. Asati[1], Raju Kota Solomon[2]

[1]Department of Electrical and Electronics Engineering, BITS-Pilani, Pilani Campus, Vidya Vihar, Pilani - 333031, Rajasthan, India.

[2]Reconfigurable Computing Systems & Wireless Sensor Network Systems Lab, Digital System Group, CSIR-CEERI, Pilani - 333031, Rajasthan, India; ashishmishra@pilani.bits-pilani.ac.in, achhu.05@gmail.com

Abstract

Validation of the robustness, efficiency of allocation and scheduling heuristics in large scale parallel and distributed systems is usually done using synthetic randomly generated workloads, represented by task graphs. Randomly generated graph are required for verification of algorithms in multidisciplinary streams. This requires that the number of nodes and the connections can be large ranging from few nodes to thousand nodes, which is demands the machine assisted development. These graphs are used as input format for many domains and require the simplest format for parsing. This research work focuses on generation of such graphs in IBM Graphviz dot format by defining the user requirement in simple formats. Three algorithms have been proposed which generate graph with proper inter connections. The task nodes are placement randomly using a layer-by-layer approach and then connected randomly. The developed generator called Modular Random Task Graph generator (MRTG) can generate task sets containing several different types of task graphs like rooted trees, isomorphic graphs and similar graphs with same node placement but different connections, with the flexibility to dictate the type of graph generated. The developed tool allows the user to generate simulated input and can be extended to any format as it is written in modular format in C++.

Keywords: Graph Generation, Hardware Software Codesign, Isomorphism, Task Graph

I. Introduction

Research in embedded real-time systems, operating systems and hardware software co-design, as well as in more general allocation and scheduling fields, is hampered by the lack of a common base of examples. In general, an example used in allocation and scheduling research consists of a task set and a database of processors and communication resources. A task set is a collection of task graphs, each of which is a directed acyclic graph (DAG) of communicating tasks. Generation of sample task sets is often a requirement when comparing allocation or scheduling methods with each other. The existing solutions are of limited relevance in today's scheduling problems in parallel, distributed systems and fields like hardware software co-design, which require a clear definition of the size of the critical path and also need the possibility of defining different types of task nodes with independent parameters. Our work accomplishes these and also gives researchers an opportunity to clearly define the number of inputs to each type of task node, which is necessary in today's computer science scheduling problems which require that all inputs arrive for the task node to give an output.

MRTG is highly valuable for scheduling simulation in the problem of many core processors, to choose how to distribute the work load done among such large number of processing cores. It is of particular importance of researchers working in areas like reconfigurable computing and System on Chip, because of its layer-by-layer[1] approach, as researchers can define reconfiguration in-between different levels. As MRTG has been created with

*Author for correspondence

computer hardware scheduling problems in mind, the constraints are defined in terms of the silicon area consumed in executing each task and the delay across that task node. But, these definitions are flexible and we can interpret these constraints in the way that they are relevant to your research. For easy analysis, MRTG currently gives output graphs in two formats a text file with the list of node placement, an array of ordered pairs describing the connections and in GraphViz's DOT format[2].

In Section II, we present a thorough comparison of MRTG with popular existing random task graph generators such as Task Graphs for Free (TGFF)[3], Graph Generation for Scheduling Simulations (GGEN)[4] and Random Task and Resource Graphs (RTRG)[5]. In Section III, we describe the working of MRTG, which is currently divided into four modules assignLevels, connectNodes, isomorphize and plotGraph. We elaborate on the future possibilities with MRTG in Section IV, while Section V presents the conclusion.

II. Comparison with Existing Solutions

MRTG differs from all existing solutions in the respect that it is divided into self-contained modules and changes made in one module don't affect the functioning of another. This modular nature makes the code far more reusable than a conventional monolithic design. Future improvements are easier to make, as additional modules can be added without disturbing the functionality of the original stable software. It also makes the program very flexible to use, as now researchers can choose to run only those modules which they require and also change the order of execution of modules to suit their needs.

Given below are a few comparisons of MRTG with existing popular algorithms for random task graph generation.

A. TGFF: Task Graphs for Free

TGFF[2] is one of the oldest and most popular algorithms for generating user-controllable, general-purpose, pseudorandom task graphs. The original TGFF algorithm iteratively adds nodes to construct a graph using limits on the maximum in and out degrees of each node, this reduces the randomness of the task graph generated making it a pseudo-random task graph generator[3]. A more recent version provides an option to generate series-parallel DAGs also. MRTG on the other hand, keeps the node placement, connection steps separate and adds randomness at every level making it a truly random task graph generator. This method of iteratively adding nodes, leads to generation of only rooted task graphs by TGFF, while MRTG can generate graphs with any number of input nodes, including rooted trees if the number of input nodes specified by the user is one. MRTG creates a graph with the exact number of nodes as specified by the user, while TGFF takes the average and multiplier from the user, for the lower bound on the number of nodes in a graph and creates a graph with number of nodes that are randomly greater than this lower bound[3].

One major difference between TGFF and MRTG is that, TGFF uses a concept of depth to decide the communication delay of the graph generated. In MRTG we have a concept of levels, where it assumed that any task in the next level does not start unless all the tasks in the previous level are completed. This is of particular importance to researchers in fields like reconfigurable computing, as now they can define different levels in random task scheduling to take into account reconfiguration of the hardware between levels.

TGFF is flexible with the number of inputs of each node and the user specifies the degree of inputs. The TGFF algorithm results in a graph that has nodes in which the number of inputs connected is any number less the degree specified, while MRTG takes the exact number of inputs required by each task from the user and ensures that they are connected. A sample generated using TGFF is shown in Figure 1.

Even so, MRTG and TGFF are similar with respect to the flexibility of the outputs of a node as both take the

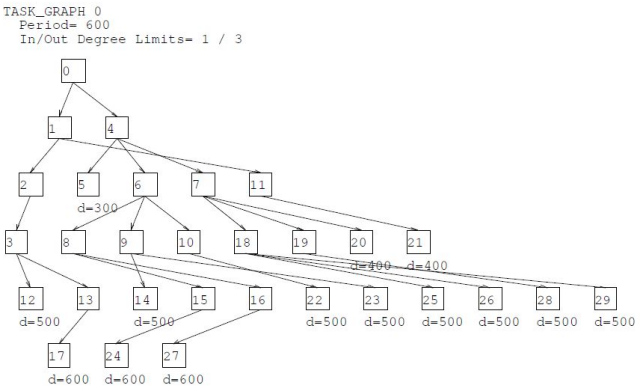

Figure 1. A graph generated using TGFF.

maximum number of outputs and connect any number below this. Also, in TGFF only one type of node is specified, while in MRTG, multiple types of nodes can be specified with their own individual constraints.

B. GGEN: Graph Generation for Scheduling Simulations

GGEN[4] provides a very thorough coverage of the different task scheduling algorithms developed over the years. It uses the libraries of the existing solutions and provides the developer with a single tool to exploit them. But, in doing so it gets limited by the shortcomings of those algorithms. It becomes a one stop tool for developers, but has limited additions of its own. Different types of nodes with their own individual parameters and constraints can be specified in MRTG, while this is not an option in GGEN.

One aspect where GGEN and MRTG are similar is that both generate an output in Graphviz's DOT language[2].

C. RTRG: Random Task and Resource Graphs

RTRG[5] is a simple and effective tool, but provides very limited flexibility to the developer. It defines resources used by a task node, which is similar to the concept of area used in MRTG. It offers different types of nodes but the constraints of each node have to be defined individually, which presents a problem when the number of nodes increases, while the node type definitions in MRTG are grouped making it easier for the user. The output file generated by RTRG is in .rtg format, which is difficult to analyze, while MRTG generates a DOT file[2] as output along with a text file showing the connections and node placement in matrix form. The work in[6] shows the verification of point for such generated graphs. Further analysis has been shown in[7].

III. Algorithmic Design of MRTG

MRTG primarily generates a specified number of random isometric task graphs, where the graph nodes are tasks and the graph edges depict the communication between tasks. In the algorithm, we first decide the total number of task nodes needed, the number of levels in which to place them, number of input nodes, the maximum area that each level can accommodate and the total delay constraint. Along with this, we can specify different types of task nodes here, by giving the number of inputs, the fan-out, i.e. degree of the output, area consumed, delay and the total number of nodes of that type. Here, we can also give the number of isomorphic graphs[8] required. Rooted graphs can also be generated by specifying the number of input nodes as one. The seed for randomness in MRTG, which decides the structure and other aspects of the generated task graphs, is the current system time making it highly unlikely for any two graphs generated at different times to be similar. But if similar task graphs are required, we can specify a user-defined seed and share it with another researcher. We have defined four rules in this algorithm:

- Every type of node has only specified number of inputs, but the output can go to any number of nodes less than the fan-out as input.
- The output of every node, except the ones the bottom level, is connected to at least one input.
- All connections are downward directional, so the output of a lower node cannot connect to an input of an upper node and any node's output can go into the input of a node from any level below its own.
- Tasks in a level start only after all the tasks in the previous level are completed.

A. Module 1 - assignLevels

This module develops on the layer by layer method proposed by Tobita and Kasahara[1]. Here, we randomly place node in different levels, without violating the maximum area that each level can accommodate and the total delay constraint. This module only decides the node placement and makes no connections between the nodes. We first create a list, which stores the node placement information. Then, we repeatedly select a random level and put a randomly selected node in it, while ensuring the maximum area in that level is not exceeded. This process continues till all nodes are placed. Lastly, we calculate the total delay of this particular node placement by adding the maximum delay at each level and check against the delay constraint. If violated, the process repeats from the start. Although very simple, this method is very useful in practice because by limiting the number of levels we can limit the size of the critical path.

Figure 1 : Module 1 - assignLevels

1. **Name: assignLevels**
2. **Function:** Randomly place node in different levels, without violating constraints
3. **Input:** numberOfLevels = The total number of levels in the graph
4. areaPerLevel = The maximum area that each level can accommodate
5. delayLimit = The delay constraint
6. inputNodes = Array containing all the input nodes
7. taskNodes = Array containing all the task nodes
8. nodeList = List defining node placement, with the first index holding the level number
9. **Output:** Passed by reference
10. Boolean value returned
11. **Algorithm:**
12. /* **assume nodes sorted already by type** */
13. Define an array freeArea[] and put the value of areaPerLevel for each level.
14. Insert all input nodes at level 0 of nodeList and subtract area of each from freeArea[0].
15. iffreeArea[0] is less than 0
16. then say "Input nodes occupy too much area" and assert false
17. endif
18.19. /* **fulfill area constraints** */
20. for i=0 to taskNodes.size()
21. Initialize chosenLevel to -1
22. Initialize numIterations to 0
23. do
24. set chosenLevel as 1 + modulus of random number with numberOfLevels
25. ifnumIterations is 1e6
26. then say "Unable to find suitable level in time" and return false
27. endif
28. whilefreeArea[chosenLevel] is less than area at taskNodes[i]
29. enddowhile
30. assert false if chosenLevel is equal to -1
31. Push back the node at taskNodes[i] into nodeList[chosenLevel].
32. Subtract the area of taskNodes[i] from freeArea[chosenLevel].
33. endfor
34.35. /* **fulfill delay constraints** */
36. Initialize totalDelay to 0
37. for i=0 to numberOfLevels
38. Initialize maxDelay to 0
39. for j=0 to size of nodeList[numberOfLevels]
40. setmaxDelay to maximum of maxDelay and delay at nodeList[numberOfLevels][j]
41. endfor
42. Add maxDelay to totalDelay
43. endfor
44. iftotalDelay is greater than delayLimit
45. then return false
46. else
47. return true
48. endif

B. MODULE 2 - connectNodes

Here, we take the node placement information from the previous level and randomly make downward directional connections and store them in an array of ordered pairs, while satisfying the rules of the algorithm.

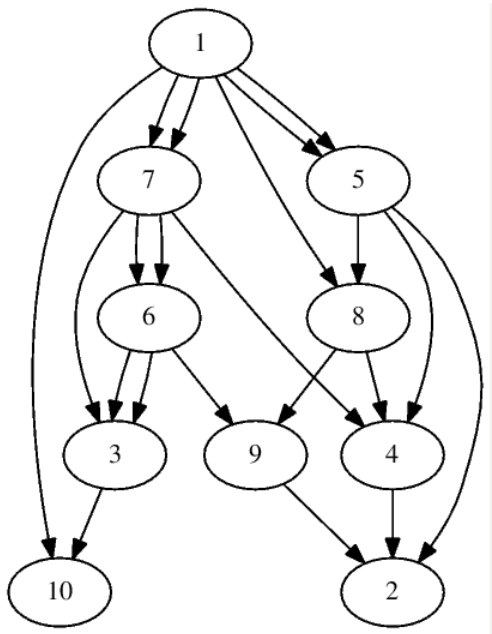

Figure 2. A rooted graph generated using MRTG.

We start moving up from the second level from the bottom and randomly connect each output only once to an input below. After all the outputs have been connected once, we start moving down from the level below the input level and randomly connect all unconnected inputs to an output above, while ensuring that the fan-out is not violated.

A totally random task graph, subject to input constraints, is generated at the end of this module.

As a fail-safe, if the algorithm gets stuck at any point and is not able to place the nodes or make connections, it will automatically show an error after trying for a hundred thousand times. A sample with rooted graph is shown in Figure 2.

Figure 2: Module 2 - connectNodes

1. **Name: connectNodes**
2. **Function:** Make random downward directed connections subject to number of inputs and fanOut of outputs of each node
3. **Input:** nodeList = List defining node placement, with the first index holding the level number
4. connections = Array of ordered pairs defining connections between nodes
5. **Output:** Passed by reference
6. Boolean value returned
7. **Algorithm:**
8. Clear any previous connections.
9. /* **ensure every input node is utilized atleast once:: bottom to top** */
10. Initialize array of ordered pairsfreeInput
11. for i=0 to size of nodeList[numberOfLevels]
12. for j=0 to number of inputPins of nodeList[numberOfLevels][i]
13. Push back (numberOfLevels,i) into freeInput
14. endfor
15. endfor
16. for level=numberOfLevels-1 to 0
17. Randomly shuffle the array freeInput
18. for i=0 to size of nodeList[level]
19. Initialize a node id to nodeList[level][i]
20. if size of freeInput is 0
21. then return false
22. endif
23. Initialize choice to freeInput.size()-1.
24. Initialize newLevel to the first value in the pair freeInput[choice]
25. Initialize indexInNewLevel to the second value in the pair freeInput[cho.ice]26
26. Initialize node newId to nodeList[newLevel][indexInNewLevel]
27. Push back (id,newId) into connections
28. Increment the value of connectedOutputs of nodeList[level][i]
29. Increment the value of usedInputPins of nodeList[newlevel][indexInNewLevel]
30. Pop back freeInput
31. endfor
32. for i=0 to size of nodeList[level]
33. for j=0 to number of inputPins of nodeList[level][i]
34. Push back (level,i) into freeInput
35. endfor
36. endfor
37. endfor
38. Initialize array of ordered pairsfreeOutput
39. for i=0 to size of nodeList[0]
40. for j = connectedOutputs of nodeList[0][i] to fanOut of nodeList[0][i]
41. Push back (0,i) into freeOutput
42. endfor
43. endfor
44. Initialize highestLevelInFreeOutput to 0
45. Sort the elements in freeInput
46.
47. /* **now process all the free inputs top to bottom** */
48. for i=0 to size of freeInput
49. Initialize level to the first value of the pair freeInput[i]
50. Initialize indexInLevel to second value of the pair freeInput[i]
51. Initialize node id to nodeList[level][indexInLevel]
52. while level - highestLevelInFreeOutput is greater than 1
53. Increment highestLevelInFreeOutput
54. if level is equal to highestLevelInFreeOutput
55. then return false
56. endif
57. for ii=0 to size of nodeList[highestLevel InFreeOutput]

58. for j= number of connected-Outputs of nodeList[highestLevelInFreeOutput][ii] to fanOut of nodeList[highestLevelInFreeOutput][ii]
59. Push back (highest Level In Free Output, ii) into freeOutput
60. endfor
61. endfor
62. Randomly shuffle the array freeOutput
63. endwhile
64. if size of freeOutput is 0
65. then say "Insufficient outout pins" and return false
66. endif
67. Initialize choice to size of freeOutput -1
68. Initialize newlevel to the first value in the pair freeOutput[choice]
69. Initialize indexInNewLevel to the second value in the pair freeOutput[choice]
70. Pop back freeOutput
71. Initialize node newId to nodeList[newlevel][indexInNewLevel]
72. Push back (newId,id) into connections
73. Increment the usedInputPins of nodeList[level][indexInLevel]
74. Increment the connectedOutputs of nodeList[newLevel][indexInNewLevel]
75. endfor
76. Return true

C. Module 3 - Isomorphize

A graph G is isomorphic to a graph H if there exists a one-to-one function, called an isomorphism, from V(G) (the vertex set of G) onto V(H) such that (u_1,v_1) is an element of E(G) (the edge set of G) if and only if (u_2,v_2) is an element of H[6]. In simpler terms, two graphs are isomorphic when the vertices of one can be re labeled to match the vertices of the other in a way that preserves adjacency. This module is used to generate graphs that are isomorphic to the one generated above. We randomly select a type of node and swap the identification numbers of any two nodes of that type. The number of times this process is repeated for each isomorphic graph is also random. A sample with two isomorphic is shown in Figure 3.

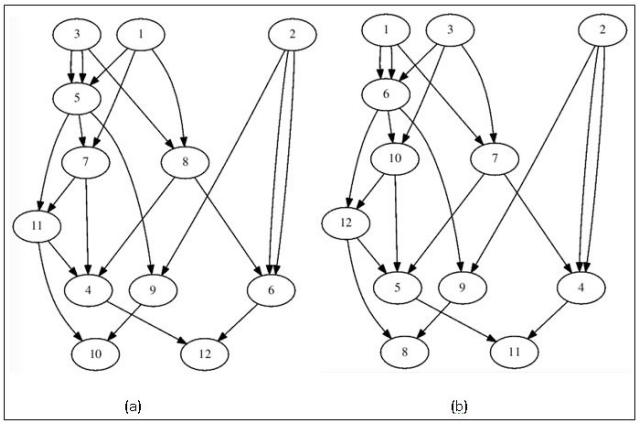

Figure 3. Two isomorphic graphs generated using MRTG.

Figure 3: Module 3 - Isomorphize

1. **Name: isomorphize**
2. **Function:** Generate isomorphic graphs
3. **Input:**typeList = List defining the type of node, with the first index holding type and the second one holding the node information.
4. **Output:** Passed by reference.
5. **Algorithm:**
6. for i=0 to size of typeList
7. while generated random number %10 is not 0
8. int u = modulus of randomly generated number with size of typeList[i];
9. int v = modulus of another randomly generated number with size of typeList[i];
10. define Node n1 as typeList[i][u] and Node n2 as typeList[i][v];
11. swap the ID of n1 and n2 by reference;
12. endwhile
13. endfor

D. Module 4 - plotGraph

The DOT language[2] provides syntax for describing graphs, edges, nodes and the properties associated with the graph components in simple text format. We have chosen Graphviz's DOT language as the default format for graph representation for MRTG, to make it compatible with most of the available tools for graph analysis. In this module, we create graphs in DOT file format[2] from the list of nodes and array of connections created above. MRTG being modular gives a lot of flexibility and control to the researcher. You can run assignLevels module once

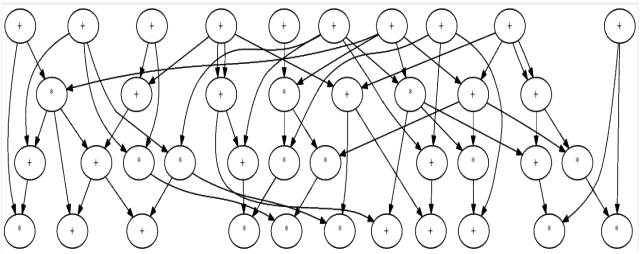

Figure 4. Nodes with operators generated using MRTG.

and connectNodes module multiple times to generate similar graphs that have the same node place but different connections between the nodes. Rooted trees can be generated by specifying only one input node. Any kind of simple modification in the generated graph can be done as shown in Figure 4 where nodes have a operators within them.

IV. Future Work

Being modular, MRTG can have future additions in the form of modules, which can be added without disturbing the original stable software. We plan to make it open source so that researchers who really need it, can develop modules they need and add them to the project so the whole community can use them. We plan to develop a module to add weights to the connections too. This will be very useful for researchers who need to do scheduling while taking into account the communication delay and resource expenditure. After that we also plan to add a concept of depth, as proposed in TGFF[3].

V. Conclusion

MRTG provides a modular approach for generating user-controlled, truly random task graphsthat find relevance in simulating today's scheduling problems in parallel, distributed systems and fields like hardware software co-design**.** This modular nature makes the program code far more reusable than a conventional monolithic design. Future improvements are easier to make, as additional modules can be added without disturbing the functionality of the original stable software. It also makes the program very flexible to use, as now researchers can choose to run only those modules that they require and also change the order of execution of modules to suit their needs.

The layer-by-layer approach followed in MRTG, with the ability to define different types of nodes with their individual parameters separates it from existing available solutions and makes it highly valuable forresearchers working in areas like reconfigurable computing, System on Chip and for scheduling simulation in the problem of many core processors, to choose how to spread the work among such large number of processing cores.

VI. References

1. Tobita T, Kasahara H. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling, Wiley. 2002; 5(5):379-394.
2. Gansner ER, North SC. An open graph visualization system and its applications to software engineering. Software: *Practice and Experience*. 2000; 30(11):1203-1233.
3. Robert PD, David LR, Wolf W. TGFF: Task Graphs for Free. Proceedings of the 6th International Workshop on Hardware/Software Co-design, 1998; 97-101.
4. Cordeiro D, Mounie G, Perarnau S, Trystram D, Vincent JM, Wagner F. Random graph generation for scheduling simulations. Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques. 2010; 60.
5. Shafik RA, Al-Hashimi BM, Chakrabarty K. Soft Error-Aware Design Optimization of Low Power and Time-Constrained Embedded Systems. Proceedings of the Design, Automation and Test in Europe, 2010; 1462-1467.
6. Geetha NK. Verification on a Given Point Set for a Cubic Plane Graph, Indian Journal of Science and Technology. 2015 July; 8(13):55032.
7. Ramachandran M, Parvathi N. The Medium Domination Number of a Jahangir Graph Jm,n. Indian Journal of Science and Technology. 2015 March; 8(5):400-406.
8. Gross Jonathan L, and Yellen Jay, eds. Handbook of graph theory. CRC press; 2003.