Scheduling Direct Acyclic Graphs on *Massively Parallel 1K-core Processors*

Ke Yue[1], Ioan Raicu[1,2]

[1]*Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA*
[2]*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA*

SUMMARY

The era of manycore computing will bring new fundamental challenges that the techniques designed for single core processors will have to be dramatically changed to support the coming wave of extreme-scale computing with thousands of cores on a single processor. Today's programming languages (e.g. C/C++, Java) are unlikely to scale to manycore levels. One approach to address such concurrency problem is to look at many-task computing (MTC). Many MTC applications are structured as graphs of discrete tasks, with explicit input and output dependencies forming the directed edges. We designed both static and dynamic schedulers for such MTC applications, scalable to 1K-cores. The simulation study by using a cycle accurate NoC simulator shows that the proposed strategy result in 85% shorter makespan and 90% higher utilization in comparison to random mapping. In addition, our heuristic can tolerate variance of the tasks' execution times at runtime and deliver improved makespan and utilization.

KEY WORDS: NoC ; Many Task Computing ; Scheduling

1. INTRODUCTION AND MOTIVATION

In recent years, chip multiprocessors(CMPs), also known as multicore or manycore processors, have been increasingly prevalent and undergo an continuous evolution. Large-scale CMPs, such as Tilear 64-core TILE64 processor [1] and Intel 80-core teraflop processor [30], have been emerging in the industry. Furthermore, Intel has tried to squeeze over a thousand processor cores onto a single chip[2], and it was predicted that thousand-core processors will be commercialized over the next decade.

In terms of communication infrastructure, the Network on Chip (NoC) [5, 9]is generally regarded as the most promising interconnect solution for gigascale Integrated Circuits(ICs) [5] such as manycore processors. Different communication traffic of the on-chip network will contribute to different performance for the sake of the traffic flow latency. In reality, different applications have different traffic flows. Even with the same application, depending on the deployment of computation units of the application on the cores, the traffic pattern can be entirely different. For homogeneous manycore systems, since the processor speeds are identical, on-chip communication latencies become a dominant factor in determining temporal behavior for an application running on the manycore system.

Connecting cores on the chip through an on-chip network has several advantages over dedicated wiring, potentially delivering high-bandwidth, low-latency, low-power communication over a flexible, modular medium. The pattern that the nodes are connected to each other varies depending on different chip interconnection fabric. Typically, the NoC network architecture on an multi-core includes several cores with routers attached to them.

*Correspondence to: kyue@hawk.iit.edu

1.1. Scheduling Technique

To realize the maximum performance potential for executing an application on an NoC based manycore platform, twofold promising methods are proposed for customizing the application-specific NoC design. The first category of methodologies is to automatically design the NoC topology [4, 20] which is tailor-made for a specific application and satisfies the communication constraints of the design. An irregular NoC topology is generated by partitioning and mapping the task graph to the cores using an automatical synthesis process which satisfies the design objectives and constraints of the targeted application. These NoCs are developed specifically for one application or as a platform for a small class applications.

Another proposed category of methodologies is to utilize scheduling algorithms [12, 32, 27] for scheduling application-specific communication and computation tasks onto standard NoC topologies, such as mesh, torus, etc. The scheduler is mostly implemented at the operating system level, which is responsible for assigning the tasks to cores aiming to maximize performance, decreasing energy cost, or reducing the thermal envelope. The scheduler has a good understanding of the traffic characteristics for the task graph and the target NoC architecture. By using this information, an application specific assignment strategy is employed automatically for maximizing the optimization goal.

This paper focuses on developing the second category scheduler which automatically decides the execution order of the computations and communication traffic flow of the task graph on general purpose manycore platforms.

1.2. MTC Applications Represented by DAG

Many-Task Computing (MTC) applications originally introduced by Raicu [25] [26] are usually linked into workflow graphs, including a number of discrete tasks with explicit input and output dependencies between the tasks. MTC applications are typically communication intensive or data intensive, impose a requirement for a larger number of computing resources over short periods of time.

The MTC applications usually involve many tasks, ranging from tens of thousands to billions of tasks, and have large variance of task execution times ranging from hundreds of milliseconds to hours. MTC have covered a wide range of domains, from astronomy, physics, neuroscience, medical imaging, chemistry, climate modeling, economics, and data analytic. Given a predefined DAG for the MTC applications with known execution time of the task node and traffic flow, and a target manycore architecture, the problem is to find the execution order of the computational tasks and the communication transactions on limited computation and communication resources, which yield the minimum execution time and utilization rate, on the target manycore architecture.

The work presented in this paper could benefit MTC systems that are dataflow driven (e.g. Swift [31] [23], Cham++ [16], Fortress [3], Chapel [6], X11 [22]); these dataflow parallel programming systems often use DAGs to represent the dataflow, making them great candidates for our proposed scheduling heuristic to schedule large scale DAGs.

1.3. Contributions

This work's contributions can be summarized as follows:

1. The design and implementation of a heuristic based static and dynamic scheduler for large DAGs.
2. The proposed scheduling heuristic could maximize the optimization goal, such as makespan and utilization, especially for data intensive applications.
3. Proposed heuristic can tolerate the variance of the task's execution time at runtime and deliver improved makespan and utilization.

1.4. Organization

The remainder part of this paper is organized as follows. We first formulate the network latency for wormhole routing in Section 2 which is necessary in the scheduling algorithm. Then , the static and dynamic schedulers proposed are illustrated in Section 3. In Section 4, we evaluate the proposed static scheduler and dynamic scheduler. Section 4 also shows our experimental results from the NIRGAM, a cycle accurate simulator, and the random task graph generator benchmark TGFF. In Section 5, related work regarding various scheduling strategies and NoC network latency for wormhole routing method is discussed. Finally, in Section 6 we make an conclusion via summarizing our main contributions and the future work.

2. ANALYSIS OF COMMUNICATION LATENCY FOR NOC

In order to design an efficient scheduling algorithm, it is crucial to first define the temporal behavior, which is affected by computation and communication factors, on an manycore system. Since the processor speed is identical to each other on an homogeneous manycore system, the computation factor is easier to evaluate than communication. Generally, the communication between cores on the NoC is based on wormhole switching technique and XY routing method [5]. For evaluating the communication latency between a pair of cores, an accurate communication latency formula is necessary for quantity the manycore system's temporal behavior.

2.1. Communication Latency Between Cores for Traffic Flow

Now, we illustrate the latency formula we used to estimate the latency of transmitting a flit between the cores. Network latency is highly dependent on the switching technique used and wormhole routing method is employed on the manycore system. For wormhole routing, the packet sent via the link is divided into a number of flits(flow control digits) for transmission, whose size usually depends on the channel width. The flit is categorized as head flit, data flit, and tail flit. The head flit including the destination core number is end first and examined upon arrival at an intermediate core. If the next core is busy, then the whole flit is stored at buffer of this node and the rest flits will go on to transfer to the core where the head flit is waiting.

Based on the communication characteristic of wormhole routing, the communication latency between a pair of cores is determined by (1) the *distance* (the number of physical hops on the path of a flow) between the pair, and (2) the *congestions* that occur on each hop caused by interferences from other traffic flows contending for the link.

Therefore, in order to estimate the effect of congestions on the communication latency, we introduce the concept of the usage count of a physical link. Intuitively, the usage count $U_{x,y}$ of a physical link $e_{x,y}$ under a specific routing algorithm is the number of traffic flows that use the link (assuming unidirectional).

Example 1 (Usage Count)
Consider the following 4×4 mesh with $16 \times (16-1) = 240$ pairwise traffic flows as shown in Figure 1. In X-Y routing, link $e_{1,5}$ will be used only for flows from source cores $\{c_0, c_1, c_2, c_3\}$ to destination cores $\{c_5, c_9, c_{13}\}$, and link $e_{5,1}$ will be used only for flows from source cores $\{c_4, \ldots, c_{15}\}$ to destination core $\{c_1\}$, respectively.

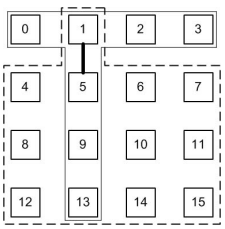


Figure 1. Usage counts of physical links $e_{1,5}$ and $e_{5,1}$

The usage count $U_{1,5}$ and $U_{5,1}$ for physical links $e_{1,5}$ and $e_{5,1}$ are respectively calculated as

$$U_{1,5} = |\{c_0, c_1, c_2, c_3\}| \times |\{c_5, c_9, c_{13}\}| = 12 \tag{1}$$

and

$$U_{5,1} = |\{c_4, \ldots, c_{15}\}| \times |\{c_1\}| = 12 \tag{2}$$

In other words, if the X-Y routing algorithm is applied, 12 communication flows use link $e_{1,5}$ and $e_{5,1}$, respectively.

In particular, according to [10], if a physical link $e_{x,y}$ has a constant processing speed, or capacity, C, i.e., then the time for a packet of M bits is deterministic with value $\delta = M/C$. Assume there is $N+1$ periodic incoming package streams (i.e., the usage count is $U_{x,y} = N+1$) with period T and packet size no larger than M, let $W_{x,y}$ be the random waiting time experienced by an arbitrary packet at physical link $e_{x,y}$ (the delay of the packet is thus $W_{x,y} + \delta$), we have the following [10]:

$$P\{W_{x,y} > t\} = T^{-N} P_N(T, t) \tag{3}$$

where

$$P_N(T, t) = \sum_{l=0}^{N-1} q_{N,l}(t)(T - N\delta + t)^l \tag{4}$$

and the coefficients $q_{N,l}(t)$ is recursively computed as

$$q_{0,l}(t) = 0 \tag{5}$$

$$q_{n,0}(t) = [\max\{0, n\delta - t\}]^n \tag{6}$$

$$q_{n,k}(t) = \frac{n}{k} \sum_{l=k-1}^{n-2} \binom{l}{k-1} \delta^{l-k+1} q_{n-1,l}(t) \tag{7}$$

for $l \leq k \leq n-1$.

Figure 2 depicts $P\{W_{x,y} > t\}$, the random waiting time of a packet at a physical link $e_{x,y}$ under various usage counts, where the common period of every flow is $T = 6$, and the processing time for a packet is $\delta = 1$. Note that the probability for zero waiting time is $P\{W_{x,y} = 0\} = 1 - P\{W_{x,y} > 0\}$.

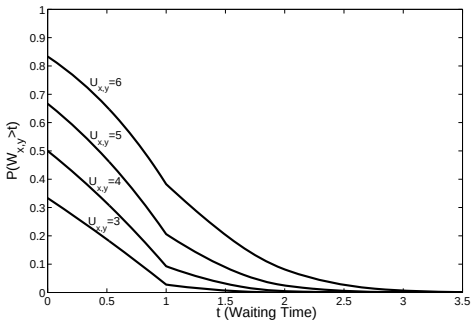


Figure 2. $P\{W_{x,y} > t\}$ under various usage count $U_{x,y}$

Equation (3) defines the waiting time of a packet at link $e_{x,y}$. As a traffic flow $\tau_{i,j}$ experience a random delay at each link along its path, the total latency $D_{i,j}$ experienced by $\tau_{i,j}$ is the sum of random variables $W_{x,y} + \delta$, $(x, y) \in \tau_{i,j}$, i.e., the random delay that occurs on each hop along the path of $\tau_{i,j}$, plus an additional delay δ of sending the packet to the router at the source core:

$$D_{i,j} = \delta + \sum_{(x,y) \in \tau_{i,j}} (W_{x,y} + \delta) \tag{8}$$

where $(x, y) \in \tau_{i,j}$ is a shorthand notation for link $e_{x,y}$ being in the path of $\tau_{i,j}$.

Note that (8) captures both the *distance* and the *congestion* factors mentioned at the beginning of this section: the distribution of each waiting time $W_{x,y}$ at link $e_{x,y}$ is determined by the usage count and thus the congestions of the link, and the summation of processing times δ along a path of a flow $\tau_{i,j}$ captures the distance. Therefore, in theory, we can obtain the probabilistic behavior of the communication latency between any pair of cores in the system based on equation (8).

However, since obtaining the distribution of the summation of random delays in (8) requires taking the convolution product of the probability measures of each random delay in (3), finding the distribution of the total latency of a flow is computationally difficult. Thus, we resort to finding the expectation of a flow $\tau_{i,j}$'s total delay $E\{D_{i,j}\}$ in Section 2.2.

2.2. *Expectations of Pairwise Communication Latencies*

Given the problem settings defined in Section 2.1, the expectation of the total latency $D_{i,j}$ experienced by $\tau_{i,j}$ is calculated as

$$E\{D_{i,j}\} = E\left\{\delta + \sum_{(x,y)\in\tau_{i,j}} (W_{x,y} + \delta)\right\} \tag{9}$$

$$= \sum_{(x,y)\in\tau_{i,j}} E\{W_{x,y}\} + (|\tau_{i,j}| + 1)\delta \tag{10}$$

$$= \sum_{(x,y)\in\tau_{i,j}} \int_0^\infty P\{W_{x,y} > t\}\mathrm{d}t + (|\tau_{i,j}| + 1)\delta \tag{11}$$

where (10) follows from the linearity of the expectation and (11) follows from the fact that for any *positive* random variable X, we have

$$E\{X\} = \int_0^\infty x f(x)\mathrm{d}x = \int_0^\infty \int_0^x f(x)\mathrm{d}y\mathrm{d}x$$
$$= \int_0^\infty \int_y^\infty f(x)\mathrm{d}x\mathrm{d}y = \int_0^\infty P(X > y)\mathrm{d}y$$

Note that in (11), $P\{W_{x,y} > t\}$ is given in (3). Although analytically calculating the integration $\int_0^\infty P\{W_{x,y} > t\}\mathrm{d}t$ in (11) is hard, the result can be obtained by numerical integration.

For instance, Figure 3 shows the relationship between the common period T and the expectation of the latency of a traffic flow of four hops in a 3×3 mesh with $\delta = 1$, $\delta = 0.8$, and $\delta = 0.5$, respectively.

As can be seen from Figure 3, the expectation of the communication latency increases dramatically as the period of every flow decreases, but approaches the congestion-free latency when the period increases.

We present the experimental results of the communication latencies in Section 2.3 and compare the analytical model presented in this section with simulations.

2.3. *Experimental Results on the Communication Latency Distributions*

In order to evaluate the correctness of the analytical model for pairwise communication latencies given in Section 2.1 and 2.2, we use NIRGAM simulator to specify a 4×4 mesh with 2 virtual channels per physical link (as in Figure 1) and generate traffic to all the other cores with injection rate[†] $\rho = 0.2$ which means each source injects a new flit one of every 5 simulator cycles. Since

[†]Injection rate is defined as the rate at which flits are injected into NoC. The simulator's cycle time is a flit cycle, the time it takes for a single flit to be injected at a source, and the injection rate is specified in flits per flit cycle.

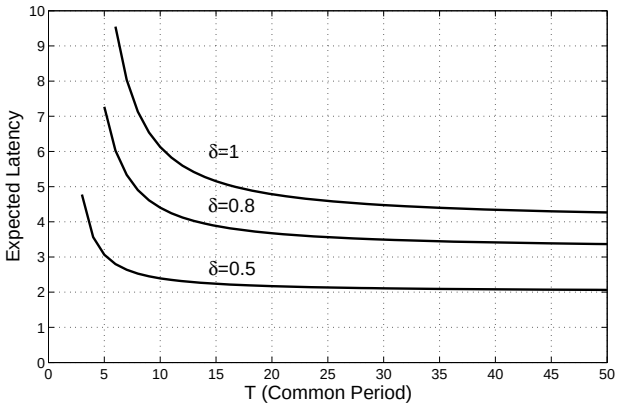
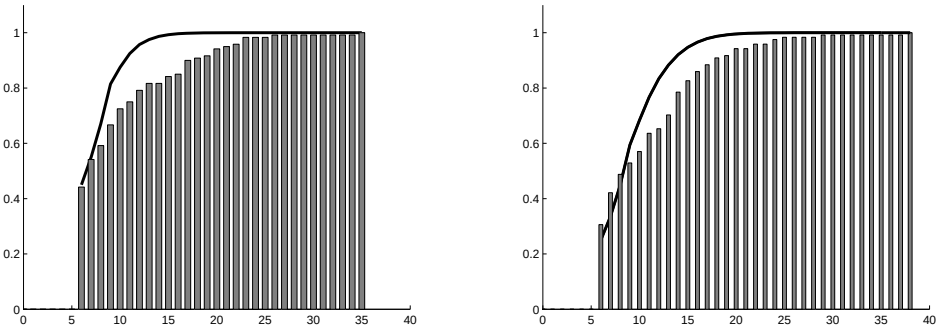


Figure 3. Expectation of Communication Latency

each core is sending flits to all the other $n-1$ cores with periods T, we have that $(n-1)\frac{1}{T} = \rho$, and thus $T = \frac{n-1}{\rho} = \frac{15}{0.2} = 75$. The router pipeline has three stages, namely, routing computation, arbitration, and crossbar traverse, in which each stage takes one clock cycle, i.e., $\delta = 3$.

With δ and T, and using the corresponding usage counts $U_{1,5} = 12$ and $U_{5,9} = 16$ for physical links $e_{1,5}$ and $e_{5,9}$, respectively, we obtain the cumulative distributions of the delays $D_{1,5} = W_{1,5} + 2\delta$ and $D_{5,9} = W_{5,9} + 2\delta$ by (3) and (8). The delay distributions are plotted in Figure 4(a) and 4(b), respectively.



(a) $D_{1,5}$ vs actual delays on $e_{1,5}$

(b) $D_{5,9}$ vs actual delays on $e_{5,9}$

Figure 4. Cumulative distributions of $D_{1,5}$ and $D_{5,9}$ of the analytical model, versus normalized cumulative histograms of delays of $e_{1,5}$ and $e_{5,9}$ in the simulation.

By comparing the theoretical distributions of the delays with the corresponding normalized cumulative histograms of delays obtained from simulation results, we conclude that the analytical model gives a good approximation of the probabilities on the lower range of communication latencies.

3. THE SCHEDULING SOLUTION

In this section, we illustrate the overall techniques for assigning the task to the processors to maximize the overall execution time and utilization rate. These include how to design an static and dynamic scheduler, as well as their execution overhead. The task scheduling problem for more than three processors is well known to be NP-complete. The methods aiming to find the optimal solution for the given objective are limited by the amount of time and memory needed since they grow as exponential function of the problem order. Because of the intractable nature of the scheduling problem, it is desirable to develop heuristic algorithms which could provide suboptimal solution within reasonable amount of computation time. Therefore, given an task graph with n nodes and manycore system with m cores, we introduce an heuristic with time complexity $O(nm)$ to get an approximate optimal scheduling result on a 1K manycore system, which features 2D mesh topology due to their structural regularity and suitability for VLSI implementation. However, the heuristic could be easily applied onto other topologies (e.g., butterfly or fat tree).

3.1. The Static Scheduler

To formulate the problem in a more formal way, we need to introduce several concepts first. For representing the characteristic of each task and traffic flows among them, the *Directed Acyclic Graph* DAG is defined as follows:

Definition 1 (Directed Acyclic Graph)
An *DAG* is a directed acyclic graph $\mathcal{G}$=G(V,E), where each vertex $v_i \in V$ represents a computational task of the application and the directed edge $e_{i,j} \in E$, represents the communication from the task v_i to v_j. The weight of the edge $w_{i,j}$ represents the data volume sent in bits from the task v_i to v_j. Task node v_i can only start to send packet after it finish its execution and its predecessor v_j can only start to execute after it receives all the packets sent from its predecessors.

Definition 2 (Task node's end timestamp)
For a given task graph, we define end timestamp of a task node v_i as v_i^e. This indicates the timepoint when the task node ends its execution.

Definition 3 (Communication cost)
For the manycore system with a DAG mapping to it, we define the communication cost between two task node v_i and v_j as C_{ij}, which is calculated as:

$$C_{ij} = \frac{w_{ij}}{M} \times E\{D_{i,j}\} \tag{12}$$

where $E\{D_{i,j}\}$ is the total latency $D_{i,j}$ experienced by traffic flow $\tau_{i,j}$ sent from the task v_i to v_j, M is the number of bits each packet has, and $\frac{w_{ij}}{M}$ is the total number of packets for traffic flow $\tau_{i,j}$

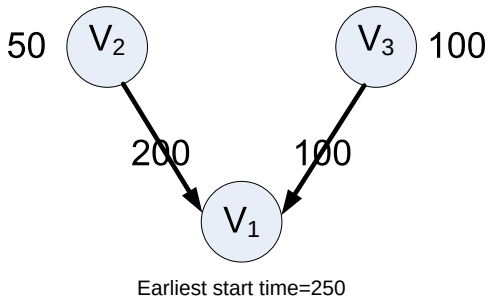


Figure 5. The earliest start time of the node

Definition 4 (Core's current time)
For the core on a chip, it may be already allocated several task nodes. The time stamp of executing all the tasks allocated on it is defined as core's current time c_i^c.

Definition 5 (Task node's earliest start time)
For a given DAG, we define task node v_i's earliest start time v_i^s as the time when the task node v_i starts to execute.

Before we start the illustration of the heuristic, it is crucial to understand the node's earliest start time v_i^s and the core's current time c_i^c. All the core's current time c_i^c is 0 initially, and will increase if task is allocated to the core. When a task is allocated to it, the core's current time will be the timepoint that it finish this task.

For the task node's earliest start time, there are generally three cases:

1. If the task node v_i has no predecessors which is usually the source node of the DAG, then the core's earliest start time v_i^s will depend on whether the core allocated to it is available. The two cases are:
   - If the core allocated to it is available and have no other task running at this time, the task could start immediately and the earliest start time v_i^s will be 0 in this case.
   - If the core allocated to it is not available, the task must wait until the timepoint, or core's current time c_i^c, when it finish the execution of the task. Thus, in this case, the task's earliest start time v_i^s would be identical to core's current time c_i^c on which it is going to run.
2. If the task node v_i has predecessors, it can only start to execute after it receive the last packet from all the predecessors. Assume we know which core these predecessors are on, then based on equation 9, we can calculate the timepoint that v_i receives the last packet sent from all its predecessors. The two cases are,
   - If the core allocated to it is ready to use, then the node v_i's earliest start time v_i^s would be the timepoint that v_i receives the last packet sent from all its predecessors.
   - If the core allocated to it is not ready to use and has tasks running on it, the task must wait until the timepoint, or core's current time c_i^c, when it finish the execution of the task. Then earliest start time v_i^s equals the core's current time c_i^c.

Take an example shown in Figure 5, the task node v_1 has two predecessors v_2 and v_3 with the corresponding end timestamp 50 and 100. The communication cost between v_2 and v_1 is 200, so that the last pack arrive v_1 at 250. Also the last packet sent from v_3 arrives at v_1 at 200 by sum v_3's end timestamp and the communication cost from v_3 to v_1. If we assume the task is assigned to core c_1, and it has finished all the execution of the task assigned to it. Thus, the task node v_1 can start to execute at time 250 which is its earliest start time.

However, if core c_1 has task running on it now and it can only finish the execution time at timepoint 300, which is just core c_1's current time c_1^c. Therefore, The task v_1's earliest start time v_1^s would be 300. Thus, based on the above example, whether one node can be executed depends on its predecessor and the core it is mapped to.

Now, based on the above analysis, we give our heuristic algorithm for a DAG with n task nodes and a manycore system with m cores as follows:

In order to give an concrete impression of our algorithm, we take an simple example illustrated in Figure 6. Assume there are several nodes in the ready list including v_2 and v_p. Sort the ready list based on their execution time and we decide to choose v_2 as the next scheduling node. v_2 has a predecessor v_1 and is already mapped to core c_6. The communication cost between v_1 and v_2 is 4. Now we start to try to schedule v_2 in the ready list since it has the smallest execution time. If we map it to the core c_1,the hop count between c_1 and c_6 is 3. Then communication cost C_{16} could be calculated by the definition.

Then we try the second core and iterate through all the cores until the last core c_m, which is 4 hop count from c_6. Choose the one with smallest C_{ij} and allocate v_2 to that core.

Algorithm 1 `Mapping algorithm`

Add the nodes which do not have any predecessors into a readylist.
Initialize the makespan as 0.
for every node in the ready list **do**
  Sort the order of the task nodes based on task node's execution time and choose the one with smallest execution time.
  for every core on the chip **do**
    Put the node on the core j and calculate node V_i's earliest start time v_i^s.
    Compare v_i^s with this core's current time c_j^c.
    If c_j^c is larger than v_i^s, then this node's earliest start time is updated with c_j^c.
    Check whether v_i^s is smallest when we map this node to different cores.
  end for
  After we find the smallest v_i^s, we map this node to the core we find.
  Update the current time on this core and this node's endtime.
  Compare this node's end time with makespan, if it is larger, then update it.
  Delete this node from the ready list.
  Check this node's successor, if its predecessors are all deleted from the readylist, add it to the readylist.
end for

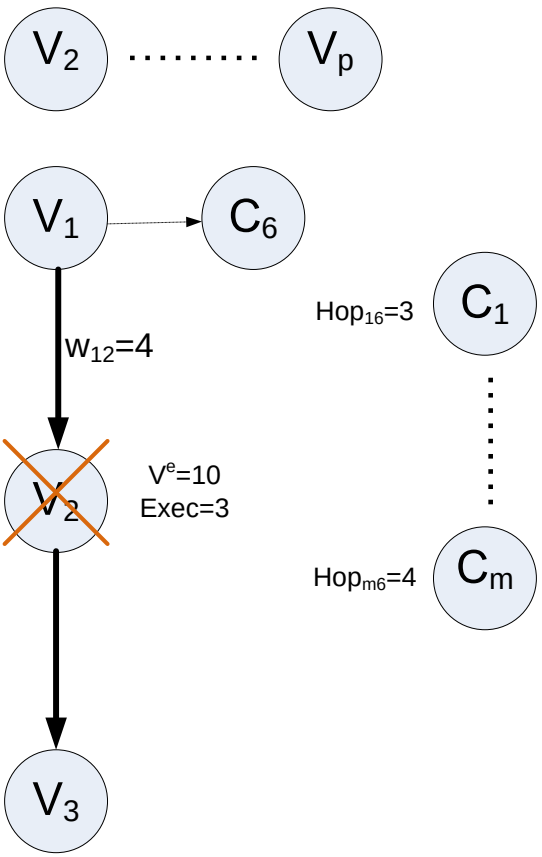


Figure 6. Simple example for the algorithm

Since v_3's only predecessor v_2 is allocated, v_3's all the predecessors are scheduled and v_3 could be added to the ready list now. And the algorithm try to start again from the first step. Figure 7 is our implementation code for the heuristic.

3.2. The Dynamic Scheduler

In this section, we define the dynamic scheduler which is based on the static scheduler introduced in the previous section. It is based on the strategy used in the static scheduler with the added flexibility of considering only a certain number of cores in making its scheduling decision. In the previous section, at each step, the candidate cores considered for scheduling are all the 1K available cores on the NoC. The time complexity to compare different candidate cores is acceptable for static scheduling, since it is executed in advance and will likely not affect the total execution time of the DAG at runtime, provided that the information of the DAG, such as the execution time of the task node and the communication time of the traffic flow, is known in advance. However, in real scenario, those information is not available until the DAG is executed on the NoC. Therefore, the time used for comparing 1K cores on the NoC for each step may affect the total execution time of the DAG on the manycore system.

To solve this issue, we set an *stepsize* parameter, which indicates the number of cores evaluated, at each scheduling step. After scheduling a task node onto one core on the NoC, we need to choose another core among the candidate cores to run the next task node in the DAG. The hop count between the candidate core and the core on which we just schedule a task node can not extend the number of stepsize. When the cores located in one corner of the mesh, the farthest core would be the one on the opposite corner. Thus for a 32×32 2D mesh with XY routing methodology, the stepsize could be ranging from 1 to 62 according the number of hops in the 2D mesh topology and XY routing method. It could be easily seen that when the stepsize equals 62, all the cores on the NoC would always be the candidate cores. In this case, the scheduling result would be identical to that get by the static scheduling.

One concern of the dynamic scheduling is the problem of computation cost which should not be omitted at runtime. In order to decrease the time spent on evaluating the appropriate cores for each task node, we further apply a storage technique. In section 2.2, it is necessary to calculate the integration $\int_0^\infty P\{W_{x,y} > t\}\mathrm{d}t$ in equation (11) by numerical integration for getting the network latency between cores. To avoid these excessive computation cost, we use a matrix to record the numerical integration result and retrieve result at runtime. For the $\int_0^\infty P\{W_{x,y} > t\}\mathrm{d}t$, the only

```
earliestT=0;
// calculate the task node's earliest start time
for (int i=0; i<readylist[0].predec.size();i++) {
    nodeindex=readylist[0].predec[i];
    xdiff=nodearr[nodeindex].x-rowindex;
    ydiff=nodearr[nodeindex].y-columnindex;
    hopcount=abs(xdiff)+abs(ydiff);
    . . .
    // calculate the earliest start time
    if (costT>earliestT){
        earliestT=costT;
    }
}
// compare the earliest start time with the core's current time
if (earliestT<corarr[rowindex][columnindex].currentTime){
    earliestT=corarr[rowindex][columnindex].currentTime;
}

// choose the core which could achieve minimum global time
Update(earliest);
```

Figure 7. Code piece

variable is T and N which decides the value for this equation. The row and column of the matrix is represented by T and N and the corresponding item in the matrix is the numerical integration value for the equation. For example, if both the value for T and N is ranging from 1 to 100K, and each item in the matrix is represented by an integer with 4 bytes. The resultant space requirement would be 40M bytes.

3.3. Run-time Overhead of the Dynamic Scheduling

For the time complexity between static and dynamic scheduling, our algorithm is based on the assumption that all the node's execution time is known a priori and is an offline static scheduling algorithm. Because we think that the comparison of the cores for each step after we choose a task node will cost much time, and this time may affect the total execution time.

However, in some cases that those time is very small compared to the node's execution time, our algorithm could be used in the dynamic scheduling algorithm. This is decided by the innate attribute of our algorithm that at every step we do not need any future information. Every information we needed, such as the predecessor's execution time, the current node's execution time, and all the core's current state are all known at this point.

Therefore, if the time spend on the scheduling is not significant, our algorithm could be used at runtime. For example, if one task node's execution is 1000 ms, while the scheduling only use 1 ms. Even there are 10^4 nodes in the task graph, the total time needed to schedule would be 10^4. And the total execution time should be at least 10^7. The effect on the makespan and the utilization rate could be acceptable under certain conditions.

Lastly, we examine the run-time of the dynamic scheduling strategy. The DAG's task node's execution time is randomly generated ranging from 1s to 50s. The communication cost on the edge will fall into the range between 10s to 20s by calculating $\frac{w_{ij}}{M}$. The computation time for the dynamic scheduling algorithm with various DAG size and stepsize is reported in Table I. The experiments were implemented on a machine with 4GB main memory and an Intel Duo 2-core CPU running at 2.2GHz. In practice, the 1K processor could allocate several processors, rather than 2 core, to work on the dynamic scheduling heuristic simultaneously, which could contribute to less computation cost.

It is worth noting that the computation cost is increasing when the stepsize parameter and the DAG size is increasing. Due to the time complexity $O(nm)$, it is expected that the computation cost for the scheduling algorithm increase linearly according to the number of task nodes in the DAG. Because m indicates the number of cores which is fixed and only the number of task nodes,n, affect the total computation time. As listed in Table I, the computation cost grows almost three time when the DAG's size is doubled.

stepsize	1k	2k	4k	8k	16k
1	0.875	3.188	10.359	43.73	142.297
2	0.86	3.203	10.344	45.297	141.08
4	0.875	3.219	10.453	44.61	143.11
8	0.937	3.313	10.765	44.39	146.125
16	1.047	3.61	11.281	45.59	148.35
32	1.391	4.297	12.766	48.469	155.98
62	2.156	5.75	15.438	54.437	164.922

Table I. The time spent in seconds of dynamic scheduling with different stepsize for DAG 1k,2k, 4k, 8k, 16k.

4. EVALUATION

In this section, we try to evaluate the performance result, such as utilization, and makespan, generated by the algorithm we proposed as well as an ad hoc random algorithm. We use Task Graphs for Free (TGFF) benchmark [7] to generate DAG with different number of task nodes and an cycle accurate NoC simulator NIRGAM [15] to evaluate the performance result. We have developed a plug-in to the NIRGAM simulator for attaching the application to NoC simulator, since the initial simulator only supports a uniform traffic pattern on the manycore topology. The simulations and TGFF benchmark are run on a machine with 4GB main memory and with an Intel Duo 2-core CPU running at 2.2GHz.

The proposed scheduling strategy result in 85% shorter makespan and in 90% higher utilization in comparison to a random mapping. We also simulate an scenario that when the execution time of a DAG changed at runtime (e.g. inaccurate DAG information), how the makespan and utilization will be affected compared with the case when the DAG is not changed. The result shows that, if the total execution time of all the task nodes in the DAG does not change and only the individual task node's execution time changed, the makespan and utilization of the proposed algorithm did not vary.

4.1. Experiment Setup with TGFF and Nirgam

TGFF is used to generate random task graphs (DAGs) whose characteristic is restricted via certain parameters in the option file. These DAGs are used in scheduling and allocation research in embedded system, operating system, as well as distributed systems. We use TGFF to generate DAGs for our experiment with size,1K, 2K, 4K, 8K, 16K. The parameter for an typical 1K DAG's option file is illustrated in Table II.

Parameter	Meaning	Value
tg_cnt	number of DAG to generate	1
task_cnt	number of task nodes	1024
task_degree	maximum number of incoming and outgoing arcs	5 6
table_cnt	number of table defining node and transmission's execution time	1
table_attrib cost	indicate cost as only variable and the average value is $80+/-20$	80 20

Table II. The parameter used in TGFF

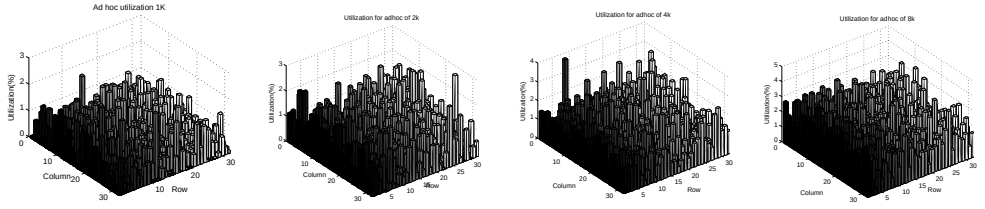
NIRGAM is a modular and cycle accurate simulator developed using SystemC. In NIRGAM, a 2D NoC mesh of tiles can be simulated by different design options, e.g., virtual channels, clock frequency, buffer parameters, routing mechanisms and applications patterns, etc. Each NIRGAM tile consists of various components, such as input channel controller, virtual channel allocator, output channel controller, and IPcores. Each IPcore is attached to a router by means of a bidirectional core channel. In NIRGAM, each tile is a traffic generator sending packets for certain time interval based on a specified task graph. Every packet is sent via wormhole switching and deterministic XY routing on the NoC.

However, the NIRGAM simulator was only designed for a maximum of 81 cores, (9×9). In order to run the simulation for massive number of cores, we have extended the simulator to support massive number cores, with a 2D 32×32 mesh. To start the experiment, the configuration file is illustrated in Table III

Since the simulator does not support traffic pattern as in the DAG's edge, we need to write program to read the information of the DAG and simulate the traffic pattern while we execute our scheduling strategy. For each tile on NIRGAM, we implement a program which could send and receive the packets attaching to each tile. The send thread send the flits to the destination tile for each certain interval based on the edge's weight of the DAG. The time interval between flits is fixed

Parameter	Meaning	Value
TOPOLOGY	manycore topology	MESH
NUMROWS	number of rows	32
NUMCOLS	number of columns	32
RTALGO	the routing algorithm	XY
WARMUP	time used to warmup the simulator	5
FLITSIZE	the size of one flit	124 bytes

Table III. The parameters used in NIRGAM



(a) Utilization for 1K with ad hoc
(b) Utilization for 2K with ad hoc
(c) Utilization for 4K with ad hoc
(d) Utilization for 8K with ad hoc
(e) Utilization for 16K with ad hoc
(f) Utilization for 1K with static scheduling
(g) Utilization for 2K with static scheduling
(h) Utilization for 4K with static scheduling
(i) Utilization for 8K with static scheduling
(j) Utilization for 16K with static scheduling
(k) Utilization for 1K with dynamic scheduling of hop 8
(l) Utilization for 2K with dynamic scheduling of hop 8
(m) Utilization for 4K with dynamic scheduling of hop 8
(n) Utilization for 8K with scheduling of hop 8
(o) Utilization for 16K with dynamic scheduling of hop 8

Figure 8. Comparison between ad hoc and scheduling utilization result

clock cycles which is defined as 1ns in our configuration file. When sending the packet, the sending timestamp, task node sending the flit, as well as the core number for the source tile is recorded in the payload of the flits. The receive thread receive these flits, record the receiving timestamp and decide whether to execute the descendent of the task node running on the source tile. When the destination tile receives the last flit of the task node executing on the source tile, it triggers its own send thread to execute the descendent task and send flits to other tile.

In the following section, we use the NoC simulator NIRGAM, as well and our plugin to evaluate the efficiency of our result generated by scheduling algorithm. After the scheduling algorithm is run, each core on the chip is allocated task nodes and execute them based on the DAG given. The DAGs are randomly generated with TGFF for 1K, 2K, 4K, 8K, 16K number of task nodes, which are mapped to a manycore NoC with 1K number of cores. The DAGs varies significantly in their size and structures, which could be used to access the quality of our heuristic with diverse abstract graphs.

4.2. Static Scheduling Evaluation

We first evaluate the quality of static scheduling, in terms of makespan and utilization rate, compared with an ad hoc scheduling strategy. The DAG's size varies from 1K to 16K, and we assume that the DAG information, including the task node's execution time, the traffic flow's transmission time, as well as the dependency relationship among all the task nodes, are known a priori before it is executed. Thus, based on the DAG, we could use the scheduling algorithm to decide the mapping strategy in advance.

Based on the scheduling strategy, we use NIRGAM to simulate the traffic flow characterized by the DAG and measure the makespan for DAG with different size. The result is shown in Figure 9, in comparison to the makespan get by an ad hoc scheduling strategy. The x axis shows the number of nodes in the task graph and the y axis shows the makespan obtained by NIRGAM. There are two lines in the Figure 9 which illustrate the makespan for two scheduling strategy. The top line with larger makespan for all the possible DAGs are obtained by running the ad hoc heuristic. The bottom line is the makespan calculated using the heuristic proposed in previous section.

For different DAGs with significantly different sizes, the makespan obtained by our heuristic always perform better than the ad hoc strategy. As illustrated in the figure, our approach achieves as high as 85% less makespan than the ad hoc heuristic. Furthermore, with the increasing of the DAG size from 1K to 16K, we could observed from Figure 9 that the difference between the two heuristic becomes larger, which means that our heuristic also behaves better on scalability. The increase of the makespan is less than that of ad hoc strategy.

Then we look at the scalability of our static scheduling methods. As plotted in Figure 9, with the increase of the DAG size, there is no steep increase in the makespan. Although the size of the DAG increased 100% at each point, the makespan is increased by 25%. Therefore, for applications with large scale DAG, performance degradation would not been an issue with the proposed static heuristic.

Now we make a comparison between two scheduling strategy in respective of the utilization rate. This is could be observed in Figure 8, where x axis and y axis show the coordination of the core on the 2D mesh. Each bar represents the corresponding core's utilization rate. As shown in Figure 8, the utilization rate calculated using our heuristic is mostly around 40%, while the highest utilization rate ad hoc strategy is only 4%.

As plotted in Figure 8, our scheduling algorithm performs better on utilization rate in addition to makespan. While the utilization rate of our figure are as high as 50%, the utilization rate of the ad hoc methods are only as high as 4%

When we increase the size of the DAG by adding more task nodes from 1K to 16K, the utilization rate for our heuristic slightly decrease from 45% to 38%. Though utilization rate of ad hoc increase from 2% to 6%, the average utilization rate of our scheduling strategy should still performs better. Because the DAG used is the same for both scheduling strategy, then the total execution time of all the task nodes would be the same, which means the average utilization rate, calculate by dividing the makespan by the average execution time of each task node, obtained by our heuristic still behaves

better than the ad hoc strategy. Thus, only a small portion of the cores behaves better on the increased percentage over the utilization rate. And the work load among the cores for our heuristic is more even when the DAG increased, which could explain why the utilization rate decrease.

That means if the computation intensive application is mapped to this platform, the algorithm could get a mapping result with better utilization rates. In order to evaluate this, we generate four different DAG with 4k task nodes. Initially ,the execution time is randomly generated from 1 to 50 and the communication cost is randomly generated from 100 to 200. Afterwards, we divided the range for the communication cost by 10 for each step, thus generating the DAG with the characteristic for the data intensive applications.

Then we apply the static scheduling strategy and generate the average utilization rate and the maximum utilization rate which is illustrated in Figure 10. The average utilization rate is the summation of the total utilization rate divided by the total number of cores. The maximum utilization rate is the highest utilization rate among all the cores on NoC. We can observe from the Figure 10 that, with the increasing ratio of the computation load in the DAG, both the average utilization rate and the maximum utilization rate start to increase.

4.3. Result for Dynamic Scheduling

In this section we illustrate the performance result for dynamic scheduling with different step size. Figure 8 shows the makespan obtained by NIRGAM simulation result for DAGs ranging from 1k to 16K, represented by 5 different lines. The x axis of Figure 8 indicates the step size, that is the total hop count number, used by the dynamic scheduling. When the stepsize equals 62, the dynamic scheduling could be deemed equally to static scheduling. The top line in the Figure 11 has the highest makespan, since it has the largest number of task nodes, which is 16K in the DAG. For all the DAGs ranging from 1k to 16k, the makespan decreased when the stepsize increase from 1 to 62.

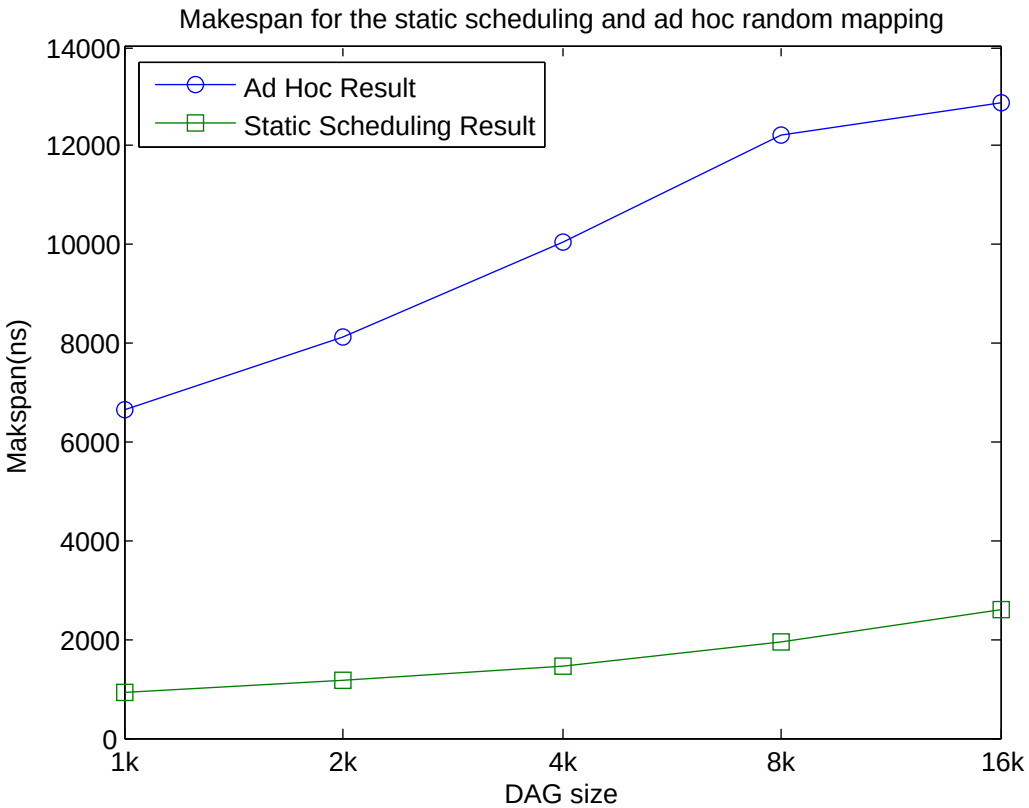


Figure 9. Comparison of makespan between static scheduling and ad hoc mapping

The reason is that, with more cores considered at each step, it is capable of getting better optimal solution from the candidate cores. But that also means spending more time for the algorithm to make scheduling strategy. In other words, it is better to choose the best stepsize which could maintain best performance result using least time. As plotted in the Figure 11, we could observe that after stepsize 16, the makespan rarely make large difference. If we choose the start cores in the middle of the 2D mesh with coordination $(15, 15)$, stepsize 16 will guarantee to cover all the available cores on the NoC. If the start core locates in the leftmost or rightmost corner, the stepsize 16 will guarantee to cover 25% cores on the NoC. Illustrated from the Figure 11, stepsize 8 or 16 could be an good option when implementing the dynamic scheduling.

So when the stepsize is 1, it is worst case for the dynamic scheduling algorithm and the makespan should be the highest. But if we compare the makespan in the Figure 9 and the Figure 11, we could see that the makespan is still less than that for ad hoc strategy. For example, the worst makespan for 16K DAG using the dynamic scheduling is 30% less than that of the ad hoc scheduling. Further, for 1K DAG, the makespan using the dynamic scheduling with stepsize is more than 50% less than the ad hoc scheduling.

If we fix the stepsize, the utilization rate for the simulation result of dynamic scheduling also performs better than ad hoc, as static scheduling, as plotted in Figure 8. In Figure 8, we use stepsize 8 and vary the DAG size from 1k to 16k. The resultant average utilization rate behave still better by our same discussion on the static scheduling algorithm. With the increasing of the DAG size, say 16k, nearly all the cores are used on the NoC.

In addition, another advantage of our algorithm could be claimed from the Figure 8. It is obvious that the cores we used are always clustered instead scattered on the NoC. This could leave a whole

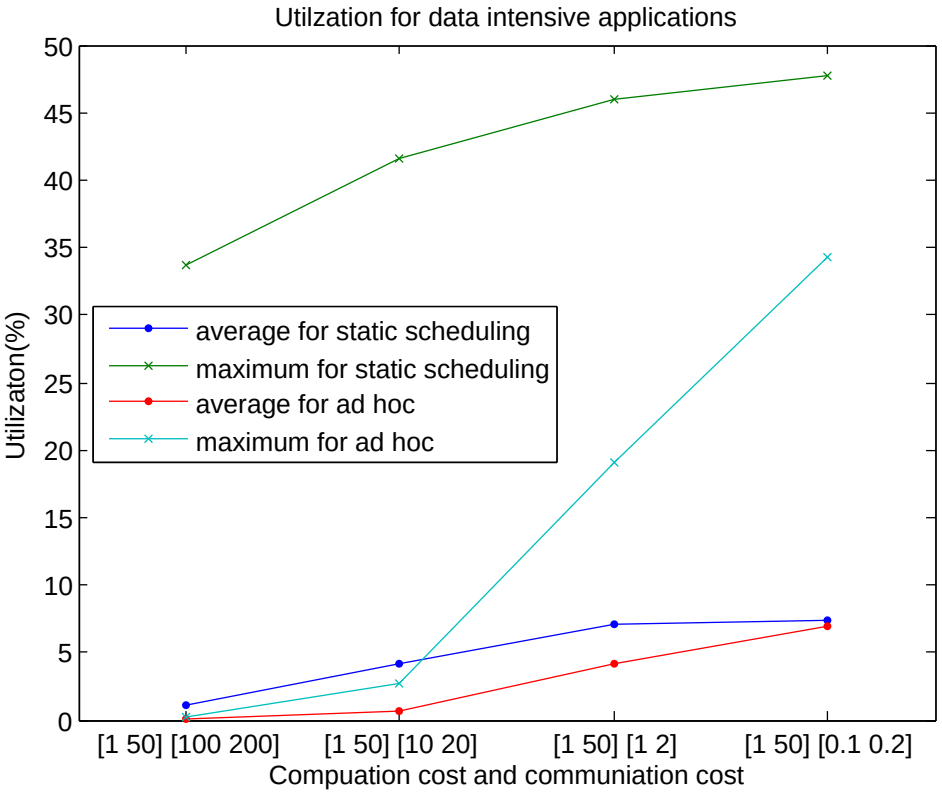


Figure 10. Average utilization and maximum utilization for communication intensive applications and data intensive applications

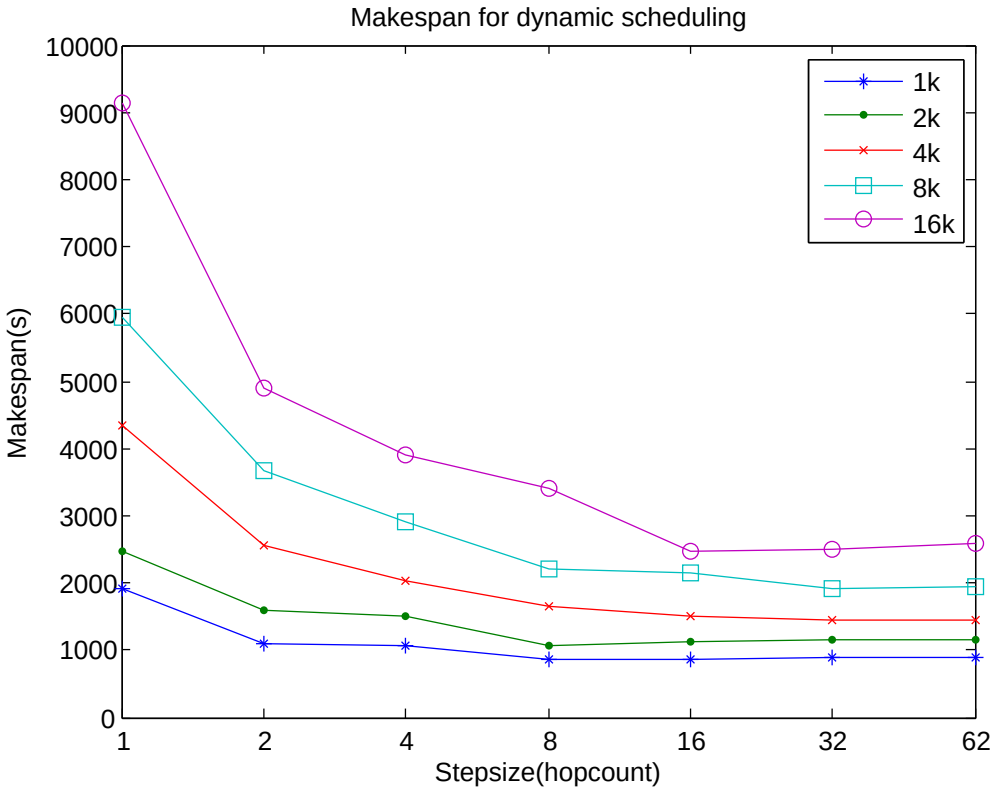


Figure 11. The makespan of different DAG for dynamic scheduling with different hop ranging from 1 to 62

area, while reducing the fragment, for being allocated to other DAGs, and improve data locality in running data-intensive DAGs.

For massive number of cores, hopcount becomes an essential characteristic that must be taken into consideration. Remote nodes tend to have less number of cores on it because the communication cost added by the hopcount.That explains why remote cores have less number of task nodes allocated on it. Because if we map the first node onto cores $(0, 0)$ at leftmost corner with the scheduling heuristic, the following nodes will be mapped to remote cores located in the direction for the rightmost corner in Figure 8. If it is put on a remote core, the hop count, or the communication cost needs to be taken into account, so the rightmost corner must not have as more task nodes allocated onto it as the cores distributed on leftmost corner to reach a smaller makespan. If we still put a lot of computations on the remote cores, the total cost of the computation and communication altogether would degrade the performance of the overall system.

4.4. Vary the Execution Time

For static scheduling, the information of the DAG, such as the task node's execution time is known a priori. However, in real system, the task node's execution may vary at runtime, thus make a possibility on performance degradation. In order to evaluate the negative effect on the static scheduling on the original DAG, we design an experiment to evaluate such result by varying the execution time at runtime.

Initially, an error rate error% is defined as the inaccuracy parameter and we implement a function generating random number according to uniform distribution between (1-error%) and (1+error%), with the average being 1. The newly generated random number is multiplied to the execution time of each node. For example, if an error rate 50% is used, the node's execution time for the new DAG would be some random number between 10*(0.9) and 10*(1.1), or between 9 and 11. We call the

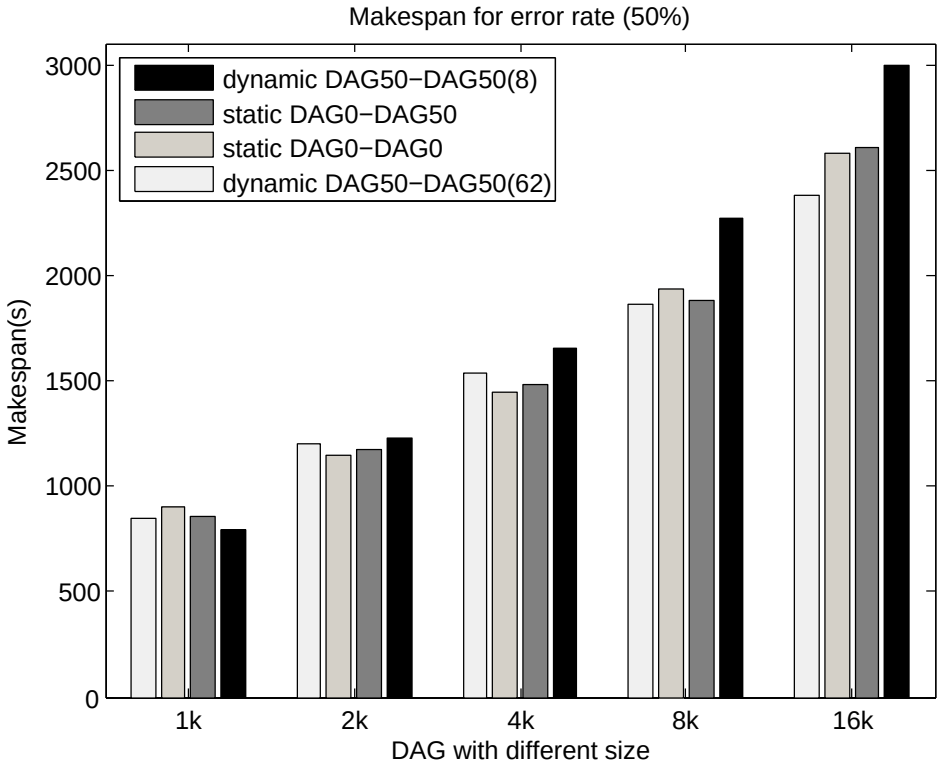


Figure 12. The makespan for error rate 50%

original DAG as DAG0 and the new DAG as DAG50. If there are large number of nodes in one DAG, the summation of the overall execution time of the different DAG equals each other because of the uniform distribution. This avoid the unfair case that the total execution time changed dramatically.

The first experiment we do is to employ our static scheduling algorithm based on DAG0, then we assume the real DAG at runtime has an errorate of 50%. In other words, DAG50 is executed based on the scheduling strategy of DAG0 indicates by the bar with notation static DAG0-DAG50 in Figure 12. The bar with notation dynamic DAG50-DAG50 means that we employ dynamic scheduling on DAG50 and still execute DAG50. The number in the parenthesis indicates the stepsize used in the dynamic scheduling. If could be seen from the Figure 12, even the individual node in the DAG may change its execution time as much as 50%, the makespan is almost identical to the result when the DAG is correct or we directly dynamic scheduling this DAG. Furthermore, even the DAG varies at runtime as much as 100%, the makespan is still within almost 5% using static scheduling strategy on DAG0 and dynamic scheduling on DAG100 as shown in Figure 13. Therefore, we could make a conclusion that our heuristic could tolerant certain degree of task execution time variance and still believed an improved performance result.

5. RELATED WORK

Before designing scheduling algorithm, it is crucial to understand the network latency encountered by each traffic flow on the NoC. Kiasari [18, 21, 19]compute the average delay due to path contention, virtual channel and crossbar switch arbitration using a queuing-based approach, which could accurately depict the blocking phenomena of wormhole switching. They calculate the delay latency using M/G/1 queuing model to predict the average message latency of a wormhole switched 2D torus NoC with deterministic routing(XY routing). Traffic model evaluating the routing delay

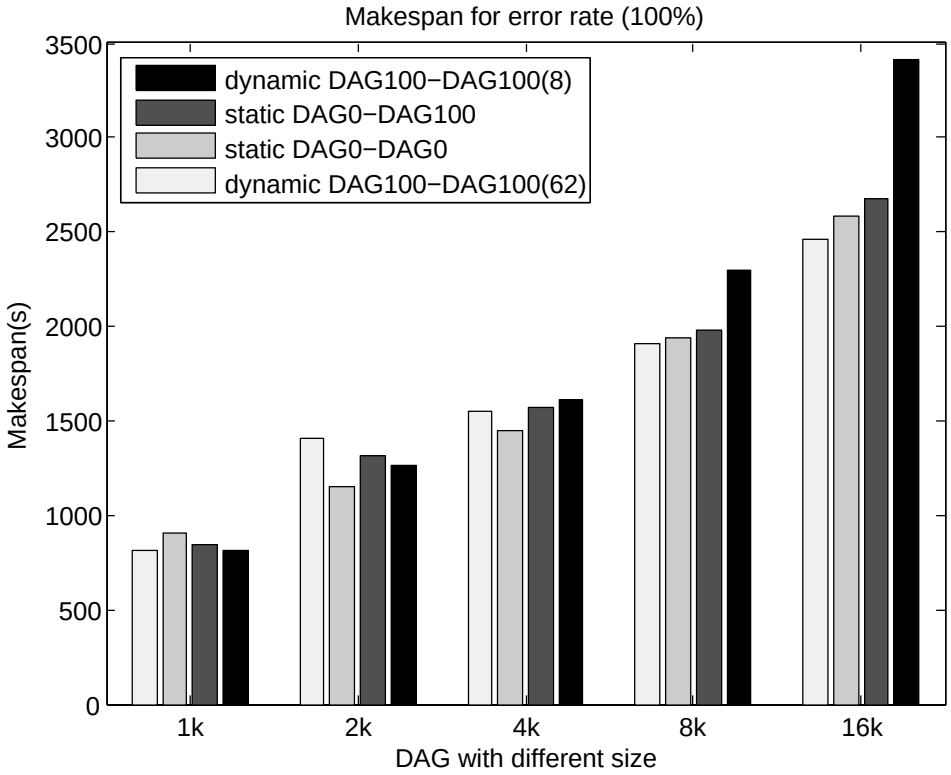


Figure 13. The makespan for error rate 100%

in heypercubes and meshes is investigated in Guan [11]. Adaptive routing performance model is built up in [8] to evaluate average latency delay in wormhole routing encompassing the switching blocking time. All the above analysis is based on assumption of the uniform traffic. Azad [28] provides a fully-adaptive wormhole-routed k-ary n-cubes latency analytical model under a non-uniform traffic pattern. Then they analysis communication delay for the bit-reversal non-uniform traffic pattern on the manycore system.

Many static scheduling approach fully exploits application-specific information off-line [32, 27, 12] for the manycore platform, which in turn leads to the performance maximization and reduce the overhead to run mapping algorithm on-line. These approaches assume the knowledge of the task graph and its characteristics before execution starts. Application-specific custom topology design has been explored in many works [4, 20, 24, 29]. By parameter tuning (e.g. buffer depth, clock frequency), instantiation, synthesis, simulation/emulation, an application-specific topology is generated.

To achieve the desired performance for applications in NoC-based manycore platform, there are generally two approaches, i.e., synthesis and scheduling. The first approach uses automatic synthesis process to partitioning and mapping a given application's task graph to cores so that the design objectives and constraints of the targeted application are satisfied [4, 20]. From initial specification, the synthesis process usually takes several hours, or even several weeks [20], to produce an optimal application-specific NoC layout, which may not necessarily be a regular mesh.

The second approach aims at general purpose NoC platform, such as mesh, torus, and uses scheduling algorithms [12, 32, 27] to map application computation and communicating tasks onto such NoC topologies. Depending on the applications' characteristics and communication cost, it is possible that not all the cores on a given platform are utilized by the application.

The above two are both up-down approaches, that first analyze applications' specifications and generate a specific hardware or a specific allocation. The topology virtualization is different from the

above, which takes a bottom-up approach. It means that the application scheduling is supposed to be fixed and developers are unaware of the hardware change. Virtualization provide a unified hardware interface for applications. The topology virtualization problem for *general purpose computing* is discussed thoroughly in [34, 33].

Different from the above scheduling methods, we focus on scheduling the large scale DAGs. The size of the DAG, on which those previous work mentioned above are working on ,is usually within several hundred. A large number of DAG should be able to saturate the communication component or impose a heavy load on the computational component. Furthermore, the DAG considered in the previous work, has an innate attribute from the DAG we considered. The DAG we studied here is actually the workflow graph, generally used in MTC, where each node stops execution after it finishes its data transmission. Previous work, especially in realtime computing literatures, focused on the DAG which keep sending packets for a certain interval and stops only when the system stops executing. Further, previous work did not consider a manycore platform with such a massive number of cores, such as 1K on a single tile. A massive number of cores impose a requirement on taking into consideration both the communication cost and computation cost at the same time. Thus, the historical scheduling method may not hold anymore for large scale DAGs mapping to manycore platform with massive number of cores.

6. CONCLUSION

Through this paper, we have successfully developed an efficient heuristic to schedule the application onto the manycore platform statically or dynamically. For massive number of cores, hop count becomes an essential effect and thus require us to consider the network latency on the manycore system. We first formulate a network latency model for quantity the communication cost encountered by the traffic flow on the NoC. Then we design and implement a heuristic based static and dynamic scheduler for large DAGs. Further, we use an cycle accurate simulator NIRGAM to evaluate the scheduling strategy. With the proposed static and dynamic scheduler, the performance metric, such as makespan and utilization, are maximally optimized, especially for data intensive applications. Furthermore, the introduced heuristic could believer improved makespan and utilization rate, when the DAG's information is not correct or the task's execution time in the DAG varies at runtime.

In future, the heuristic would take into other optimization objectives, such as energy cost, and thermal effect on the NoC. In addition, we will also strive to explore work stealing, a distributed load balancing technique that has been used successfully in parallel languages, to load balance threads on shared memory parallel machines.

We have already developed a discrete event simulator that implements the work stealing algorithm in the SimMatrix project [17]. Through SimMatrix, we showed that work stealing can be used to schedule tasks efficiently across 2D-mesh and 3D-Torus interconnects on many-core computing architectures, as well as large scale distributed systems (up to exascales). We will take the lessons learned from SimMatrix and extend the proposed scheduling algorithm from this paper to achieve distributed scheduling that could perform well in an online setting with acceptable overheads. We have also begun exploring the real implementation of the proposed scheduling algorithms on today's many-core processors, namely NVIDIA GPUs and Intel Xeon Phi accelerators, through the GeMTC project [13, 14]. The GeMTC framework supports the efficient execution of bag-of-tasks many-task computing workloads on 3K-core GPUs and 240-HT Xeon Phi. In the future, we will explore the potential of running complex DAGs directly on the GPUs and Xeon Phi accelerators at an extremely fine granularity compared to what can be achieved through the Swift workflow system.

REFERENCES

1. http://www.tilera.com/products/processors.
2. http://www.fastcompany.com/1714174/1000-core-cpu-achieved-your-future-desktop-will-be-a-supercomputer.

3. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.W. Maessen, S. Ryu, G.L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The fortress language specification. *Sun Microsystems*, 139:140, 2005.
4. L. Benini. Application specific NoC design. In — *Proceedings of the Design Automation & Test in Europe Conference*, page 105. IEEE, 2006.
5. L. Benini and G. De Micheli. *Networks on chips: Technology and Tools*. Morgan Kaufmann Pub, 2006.
6. B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
7. R.P. Dick, D.L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.
8. J.T. Draper and J. Ghosh. A comprehensive analytical model for wormhole routing in multicomputer systems. *Journal of Parallel and Distributed Computing*, 23(2):202–214, 1994.
9. J. Duato, S. Yalamanchili, and L.M. Ni. *Interconnection networks: An engineering approach*. Morgan Kaufmann, 2003.
10. A. Eckberg Jr. The single server queue with periodic arrival process and deterministic service times. *Communications, IEEE Transactions on*, 27(3):556–562, 1979.
11. W.-J. Guan, W.K. Tsai, and D. Blough. An analytical model for wormhole routing in multicomputer interconnection networks. In *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 650 –654, 13-16 1993.
12. J. Hu and R. Marculescu. Communication and task scheduling of application-specific networks-on-chip. In *Computers and Digital Techniques, IEE Proceedings-*, volume 152, pages 643–651. IET, 2005.
13. Scott J. Krieder Justin M. Wozniak Timothy Armstrong Michael Wilde Daniel S. Katz Benjamin Grimmer Ian T. Foster, Ioan Raicu. Design and evaluation of the gemtc framework for gpu-enabled many-task computing. *ACM HPDC*, 2014.
14. Anupam Rajendran Ioan Raicu. Matrix: Many-task computing execution fabric for extreme scales. *Department of Computer Science, Illinois Institute of Technology, MS Thesis*, 2013.
15. L. Jain, BM Al-Hashimi, MS Gaur, V. Laxmi, and A. Narayanan. NIRGAM: a simulator for NoC interconnect routing and application modeling. *Design Automation and Test in Europe (DATE), Nice, France*, 2007.
16. L.V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.
17. Ke Wang Kevin Brandstatter, Ioan Raicu. Simmatrix: Simulator for many-task computing execution fabric at exascales. *ACM HPC*, 2013.
18. A.E. Kiasari, D. Rahmati, H. Sarbazi-Azad, and S. Hessabi. A markovian performance model for networks-on-chip. In *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pages 157 –164, 13-15 2008.
19. Jongman Kim, Dongkook Park, C. Nicopoulos, N. Vijaykrishnan, and C.R. Das. Design and analysis of an noc architecture from performance, reliability and energy perspective. In *Architecture for networking and communications systems, 2005. ANCS 2005. Symposium on*, pages 173 –182, 26-28 2005.
20. S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo. Designing application-specific networks on chips with floorplan information. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 355–362. ACM, 2006.
21. H.H. Najaf-abadi and H. Sarbazi-azad. An accurate combinatorial model for performance prediction of deterministic wormhole routing in torus multicomputer systems. In *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, pages 548 – 553, 11-13 2004.
22. J.K. Ousterhout. An x11 toolkit based on the tcl language. In *1991 Winter USENIX Conference*. Citeseer, 1991.
23. Michael Wilde Ian Foster Pete Beckman, Ioan Raicu. Swift: Scalable parallel scripting for scientific computing. *SciDAC Review*, pages 38–53, Spring 2010.
24. A. Pinto, L.P. Carloni, and A.L. Sangiovanni-Vincentelli. Efficient synthesis of networks on chip. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 146 – 150, 2003.
25. I. Raicu, I.T. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11. IEEE, 2008.
26. Ian Foster Mike Wilde Zhao Zhang Kamil Iskra Peter Beckman Yong Zhao et al. Raicu, Ioan. Middleware support for many-task computing. *Cluster Computing*, 13(3):291–314, 2010.
27. M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 3–8. European Design and Automation Association, 2006.
28. H. Sarbazi-Azad, M. Ould-Khaoua, and L.M. Mackenzie. An analytical model of fully-adaptive wormhole-routed k-ary n-cubes in the presence of hot spot traffic. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 605 –610, 2000.
29. K. Srinivasan, K.S. Chatha, and G. Konjevod. An automated technique for topology and route generation of application specific on-chip interconnection networks. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 231–237. IEEE Computer Society, 2005.
30. S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008.
31. M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11):50–60, 2009.
32. H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for mpsoc. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 69–74. IEEE, 2009.
33. K. Yue, S. Ghalim, Z. Li, F. Lockom, S. Ren, L. Zhang, and X. Li. A greedy approach to tolerate defect cores for multimedia applications. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pages 112–119. IEEE, 2011.

34. K. Yue, F. Lockom, Z. Li, S. Ghalim, S. Ren, L. Zhang, and X. Li. Hungarian algorithm based virtualization to maintain application timing similarity for defect-tolerant noc. In *Design Automation Conference, 2012. ASP-DAC'12. Asia and South Pacific*. IEEE, 2012.