Scheduling Directed A-cyclic Task Graphs on a Bounded Set of Heterogeneous Processors Using Task Duplication

Sanjeev Baskiyar, Ph.D. & Christopher Dickinson
Department of Computer Science and Software Engineering
Auburn University, Auburn, AL 36849
baskiyar@eng.auburn.edu

Abstract

In a distributed computing environment, the schedule by which tasks are assigned to processors is critical to minimizing the overall run-time of the application. However, the problem of discovering the schedule that gives the minimum finish time is NP-Complete. This paper addresses static scheduling of a directed a-cyclic task graph (DAG) on a heterogeneous, bounded set of distributed processors to minimize the makespan. By combining several innovative techniques, including insertion-based scheduling and multiple task duplication, we present a new heuristic, known as Heterogeneous N-predecessor Decisive Path (HNDP), for scheduling directed a-cyclic weighted task graphs (DAGs) on a set of heterogeneous processors. We compare the performance of HNDP, under a range of varying input conditions, with two of the best existing heterogeneous heuristics namely HEFT and STDS. The results presented in this paper show that HNDP outperforms the two heuristics in terms of finish time and the number of processors employed over a wide range of parameters. The complexity of HNDP is $O(v^2p)$ vs. $O(v^2p)$ of HEFT and $O(v^2)$ of STDS where $v$ is the number of nodes in the DAG.

1. Introduction

The efficient scheduling of tasks is paramount to maximizing the benefits of executing an application in a distributed and parallel computing environment. The directed a-cyclic task graph (DAG) structure frequently occurs in many regular and irregular applications such as Cholesky factorization, LU decomposition, Gaussian elimination, FFT, Laplace transforms and instruction level parallelism. With widespread use of heterogeneous grid computing, such scheduling algorithms will have more frequent use. In this work our objective of scheduling DAGs is to map them onto heterogeneous distributed memory systems in such a way that the finish time is minimized, while observing precedence constraints. However, the DAG scheduling problem has been shown to be NP-Complete [7], and so much research has been done to discover a heuristic that provides best approximations in a reasonable computational period.

Past research on scheduling DAGs has explored the different combinations of the following main dimensions: (i) list and cluster scheduling (ii) bounded and unbounded processors and (iii) task duplication or not. The reader interested in a good classification of such work is referred to [24]. In the homogeneous case the state-of-art research is represented by DPS [12], DSC[19], ClusterMerge[2], and TCS [25]. This paper presents a new static list DAG scheduling heuristic for the heterogeneous environment known as Heterogeneous N-Predecessor Duplication (HNPD) to minimize makespan. It is designed for a bounded number of processors but is also shown to work well for an unlimited number of processors. After assigning priorities according to the Decisive Path, originally presented in [12], each task, in order of priority, is assigned to the processor that completes the task the earliest. Then, if there is a period of idle time on the processor, HNPD attempts to duplicate the task’s predecessors in the order of most to least
favorite (defined in Section 2) on the processor. The algorithm continues to recursively duplicate the predecessors of any duplicated tasks until no more duplication can be performed.

The remainder of the paper is organized as follows. Section 2 gives background on scheduling tasks in a heterogeneous environment, including some definitions and parameters used by HNPD. Section 3 discusses related contemporary work of other researchers. Section 4 defines the HNPD heuristic. Section 5 presents the results of the study and finally conclusions and suggestions for future work are presented in Section 6.

2. Problem Definition

To run an application on a distributed environment one can be decompose it into a set of computational tasks. These tasks may have data dependencies among them, thereby creating a partial order of precedence in which the tasks may be executed. A directed a-cyclic task graph structure occurs frequently in many important applications. A directed a-cyclic graph (DAG) has nodes in the graph that represent the tasks and the edges in the graph represent data dependencies.

In this paper, a DAG is represented by the tuple $G = (V, E, P, T, C)$, where $V$ is the set of $v$ nodes, $E$ is the set of $e$ edges between the nodes, and $P$ is a set of $p$ processors. $E(v, v_j)$ is an edge between nodes $v_i$ and $v_j$. $T$ is the set of costs $T(v, p_j)$, which represent the computation times of tasks $v_i$ on processors $p_j$. $C$ is the set of costs $C(v, v_j)$, which represent the communication cost associated with the edges $E(v, v_j)$. Since intra-processor communication is insignificant compared to inter-processor communication, $C(v_i, v_j)$ is considered to be zero if $v_i$ and $v_j$ execute on the same processor. The length of a path is defined as the sum of node and edge weights in that path.

Node $v_p$ is a predecessor of node $v_i$ if there is a directed edge originating from $v_p$ and ending at $v_i$. Likewise, node $v_s$ is a successor of node $v_i$ if there is a directed edge originating from $v_s$ and ending at $v_i$. We can further define pred($v_i$) as the set of all predecessors of $v_i$ and succ($v_i$) as the set of all successors of $v_i$. An ancestor of node $v_i$ is any node $v_p$ that is contained in pred($v_i$), or any node $v_a$ that is also an ancestor of any node $v_p$ contained in pred($v_i$).

The earliest execution start time of node $v_i$ on processor $p_j$ is represented as $EST(v_i, p_j)$. Likewise the earliest execution finish time of node $v_i$ on processor $p_j$ is represented as $EFT(v_i, p_j)$. $EST(v_i)$ and $EFT(v_i)$ represent the earliest start time upon any processor and the earliest finish time upon any processor, respectively. $T_{Available}[v_i, p_j]$ is defined as the earliest time that processor $p_j$ will be available to begin executing task $v_i$. We can mathematically define these terms as follows.

$$EST(v_i, p_j) = \max \{ T_{Available}[v_i, p_j], EFT(v_{p_k}, p_k) + C(v_{p_k}, v_i) \}$$

where, $v_p$ in $\text{pred}(v_i)$, $EFT(v_{p_k}, p_k) = EFT(v_p)$ and $C(v_{p_k}, v_i)=0$ when $k=j$.

$$EFT(v_i, p_j) = T(v_i, p_j) + EST(v_i, p_j)$$

$$EST(v_i) = \min (EST(v_i, p_j), p_j \in P)$$

$$EFT(v_i) = \min (EFT(v_i, p_j), p_j \in P)$$

The maximum clause finds the latest time that a predecessor’s data will arrive at processor $p_p$. If the predecessor finishes earliest on a processor other than $p_p$, communication cost must also be included in this time. In other words, the earliest start time of any task $v_i$ on processor $p_j$, $EST(v_i, p_j)$ is the maximum of times at which processor $p_j$ becomes available and the time at which the last message arrives from any of the predecessors of $v_i$.
The goal of HNPD is to minimize the makespan of the DAG. The makespan is defined as the time at which all nodes have finished executing. In our case, the makespan will be equal to the latest $EFT(v_i)$, where $v_i$ is an exit node in the graph.

The critical path (CP) is the longest path from an entry node to an exit node. The critical path excluding communication cost (CPEC) is the longest path from an entry node to an exit node, not including the communication cost of any edges traversed. For our problem definition, we assume that each task’s mean execution cost across all processors is used to calculate the CP. We also assume that each task’s minimum execution cost from any processor is used to calculate the CPEC.

The top distance for a given node is the longest distance from an entry node to the node, excluding the computation cost of the node itself. The bottom distance for a given node is the longest distance from the node to an exit node, including the computation cost of the node. Again, we assume that each task’s mean execution cost is used to calculate the top distance and bottom distance. The bottom distance is also referred to as the upper rank or the blevel.

The Decisive Path (DP) is defined as the top distance of a given node plus the bottom distance of the node. The DP is defined for every node in the DAG. The CP then becomes the largest DP for an exit node.

3. Related Work

List and cluster scheduling are the primary techniques to schedule DAGs. These techniques can be augmented by guided random search and/or task duplication. In list scheduling (e.g. [2 - 6], [8 - 17]) tasks are ordered in a scheduling queue according to some priority that gives the free tasks the highest priorities. Then, in order, tasks are removed from the queue and scheduled on the optimal processor. Two main design points of the list scheduling heuristic are how to build the scheduling queue and how to choose the optimal processor [1]. List scheduling algorithms have been shown to have good cost-performance trade-off [8]. Cluster scheduling involves assigning processors to nodes/paths (regardless of free or not) and merging nodes/paths to schedule on the same processor so as to get closer to the objectives of schedule length, no. of processors etc. Within the guided random search techniques, Genetic Algorithms provide good schedules, but their complexities are very high. Furthermore, the control parameters for a task graph may not give the best results for another.

There has been considerable research in scheduling DAGs to minimize finish time on homogeneous processors [2, 9, 12, 18] – a detailed survey of static scheduling algorithms for DAGs appears in [23]. The Dominant Sequence Clustering (DSC) [19] heuristic schedules DAGs yielding a finish time that is bounded within twice that of optimal for coarse grain graphs. Although there has been some research (see below) on scheduling DAGs on heterogeneous multiprocessors systems, such a bound has not been established yet.

There are several algorithms for statically scheduling DAGs on heterogeneous multiprocessor systems namely: CPOP [17], Dynamic Level Scheduling (DLS) [20], Generalized Dynamic Level (GDL) [20], Best Imaginary Level (BIL) [11], Mapping Heuristic (MH) [16], Levelized Min Time (LMT) [16], Heterogeneous Earliest Finish Time, (HEFT) [17], Task Duplication Scheduling (TDS) [14-15], Task Duplication Scheduling (STDS) [8-9], Fast Critical Path (FCP) [24] and Fast Load Balancing (FLB) [13]. Among the above, TDS and STDS employ task duplication whereas others do not. In [17] it has been shown that HEFT outperforms GDL, CPOP, MH, DLS, LMT for different CCR and $\alpha$ values (communication to computation ratio and
shape parameter of the graph, defined later). Except for FCP and FLB, the complexities of the algorithms are either equal or worse than HEFT; they are $O(v^3)$, $O(v^2p)$, $O(v^3p)$, $O(v^2p)$, and $O(v^2p^2)$ respectively where $v$ is the number of tasks in the directed a-cyclic task graph, and $p$ the number of processors. A brief description of these algorithms also appears in [17].

<table>
<thead>
<tr>
<th>Algorithm, A</th>
<th>Complexity</th>
<th>Schedule length, $L(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIL</td>
<td>$O(v^2p\log p)$</td>
<td>$L(BIL) &lt; L(GDL)$ by 20%</td>
</tr>
<tr>
<td>FCP</td>
<td>$O(v\log p + e)$</td>
<td>$L(HEFT) &lt; L(FCP)$ in general, but more so, 32% with high processor speed variance.</td>
</tr>
<tr>
<td>FLB</td>
<td>$O(v\log p + e)$</td>
<td>$L(HEFT) &lt; L(FLB)$ in general but more so, 63%, when processor speed variance is high.</td>
</tr>
<tr>
<td>HEFT</td>
<td>$O(v^2p)$</td>
<td>HEFT better than DLS, MH, LMT by 8, 16, 52% respectively.</td>
</tr>
<tr>
<td>TDS</td>
<td>$O(v^2)$</td>
<td>$L(STDS) &lt; L(TDS)$</td>
</tr>
<tr>
<td>STDS</td>
<td>$O(v^2)$</td>
<td>$L(STDS) &lt; L(BIL)$</td>
</tr>
</tbody>
</table>

The performance and complexity of the other algorithms, BIL, FCP, FLB, HEFT, TDS and STDS have been summarized in Table 1, where $e$ is the number of edges. To give an insight into the working of these algorithms we briefly discuss them below. Next, we also discuss, DPS, one of the best static DAG scheduling algorithms for homogeneous systems upon which HNPD is based.

In [11] the priority of each task is its basic imaginary makespan (BIM), which is defined using its Basic Imaginary Level (BIL). The Basic Imaginary Level (BIL) of a given task on a given processor is the length of the longest path from the given task to the exit task. The execution time of the given task on the given processor is used to calculate the longest path. The average execution times are used for the remaining tasks and the average inter-task communications are used to compute the length of the longest path. The BIL is recursively calculated (bottom-up) using the following formula $BIL(n_i, p_j) = w_{i,j} + \max_{n_k \in succ(n_i)} \{ \min(BIL(n_k, p_j), \min_{k \neq j} (BIL(n_k, p_k) + c_{k,j}) \}$. The BIM of a node $n_i$ on processor $p_j$ is computed as $BIM(n_i, p_j) = t_{av}[j] + BIL(n, p_j)$. If the number of processors be $p$, each node has $p$ BILs associated with it. If there are $k$ free tasks at any step, the priority of a task is defined as the $k^{th}$ smallest BIM value or the largest BIM value if the $k^{th}$ value is undefined. After this step, a processor is chosen for the highest priority task that minimizes its execution time. The authors state that if the number of ready tasks, $k > p$, the execution time becomes more important than the communication overhead. Therefore the BIM value is redefined as follows: $BIM^*(n_i, p_j) = BIM(n_i, p_j) + w_{i,j} \times \max(\frac{k}{p - 1}, 0)$. Tasks are scheduled onto processors on which highest revised $BIM^*$ value can be obtained.

FCP [24] reduces its complexity by restricting the processor choices for any task to the favorite processor and the processor that went idle the earliest. The task selection complexity is reduced by maintaining a list of tasks of size equal to the number of processors and sorted by decreasing blevels. Tasks with higher blevels have higher priorities. However, this reduced complexity comes at a price. In general HEFT outperforms FCP, but more so (about 30%) for irregular problems, large number of processors (16, 32 processors for 2,000 node graphs) and large heterogeneity (processor speed range varying from 4-16 units). This result holds for both low and high CCR values. FLB performance is worse, being outperformed by 63% by HEFT.
HEFT orders tasks in the scheduling queue according to their upward rank. It proceeds by first computing the \( EST(v_i, p_j) \) of all tasks on all processors. Among the free tasks, task \( v_i \) which has the highest upper rank is assigned to processor \( p_k \) which gives lowest \( EFT(v_k, p_k) \). It also explores the possibility of inserting a task in empty time slots between scheduled tasks. The \( EST \) of all successors of task \( j \) are recomputed after an assignment. It then uses an insertion-based method, similar to HNPD (explained in Section 4) to choose the processor that gives the \( EFT \) for each task. \textit{We note that HEFT can be improved by duplicating task executions on processors to reduce communication overhead.} The time complexity of HEFT is \( O(v^2 \# p) \). In CPOP [17] all tasks in the critical path of the DAG are assigned to the fastest processor and the remaining are prioritized by the sum of upper and lower ranks and mapped according to the HEFT algorithm. CPOP does not employ task duplication. However, as mentioned before, HEFT outperforms CPOP.

TDS defines for any task \( v_i \): (i) a favorite predecessor \( v_j = fpred(v_i) \) which is \( v_i \)'s last predecessor to send data (ii) the most favorite processor \( fproc(v_i) \), which is one that provides the minimum \( eff(v_i) \). The first step of TDS is a clustering strategy. A task list, sorted in ascending \( blevels \) is created. A task, \( v_i \), in the task list with the least \( blevel \) is mapped to a separate cluster \( C_i \), which is scheduled on the most favorite processor of \( v_i \), among the free processors. Some predecessors of \( i \) are added to \( C_i \) as follows. If \( fpred(v_i) \) is the only predecessor or is critical and unassigned, it is assigned to \( C_i \). Otherwise, a predecessor of \( v_i \) having the least execution time on the current processor is added to the cluster. The above steps are repeated until all tasks of the DAG are mapped onto an initial set of clusters. The next step uses task duplication to avoid communication. In the longest cluster, if predecessor \( v_j \) of task \( v_i \) is not the favorite one, all the tasks scheduled within the cluster that are prior to \( v_j \) including \( v_i \) are replaced by the favorite task \( v_k \) and all its predecessors within the cluster having \( v_k \). The replaced tasks are scheduled on a new processor. \textit{We note that TDS uses task duplication if the favorite predecessor is not in a cluster. But, replacing all tasks in the cluster that are scheduled prior to the favorite predecessor can actually increase the inter-processor communication.} TDS outperforms BIL for CCRs 0.2-1. It has a complexity of \( O(v^2) \).

STDS is an improved version of TDS as \( L(STDS) \leq L(TDS) \) in all cases. It constructs a scheduling queue of tasks with priorities based on \( level \), calculated similar to \( bottom distance \) in Section 2, except communication costs are not included. A set of values is calculated for each node \( v_i \), including favorite predecessor, which is defined as the predecessor \( v_j \) that produces the largest value of \( EFT(v_j) + C(v_j, v_i) \). It then chooses the first unscheduled task in the queue and creates a new cluster with it on the processor that yields the smallest \( EFT \). It continues to add tasks to the cluster by selecting the favorite predecessor, if critical, or the predecessor that executes quickest on the processor. The clustering continues until reaching an entry node in the DAG. Once the initial set of clusters is created, it either merges clusters or attempts task duplication. If there are more clusters than available processors, clusters are merged. Otherwise, task duplication is performed. The clusters are merged by calculating a value \( exec \) for each cluster, where \( exec \) is equal to the total of all task computations assigned to the processor. Clusters are then ordered according to \( exec \). The cluster on a real processor with the largest \( exec \) value is merged with the cluster on a pseudo-processor with the smallest \( exec \) value. To merge, tasks are ordered on the real processor according to their \( level \) value (as calculated above). This merging continues until all clusters exist only on real processors. \textit{We note that merging based upon exec values does not target minimization of makespan.} STDS performs task duplication as follows. For any task \( v_i \) in a cluster that is not preceded by its favorite predecessor, STDS takes the tail of the cluster after task \( v_i \) and moves it to the free processor yielding the \( EFT \) for the first task in the new cluster. The favorite predecessor is then recursively duplicated starting with \( v_i \) until an entry node is reached. After duplicating, the \( makespan \) is recalculated, and if it does not
decrease, the duplication is discarded. STDS has a time complexity of $O(v^2)$. Although STDS works with a bounded set of processors, it needs a large number of them to perform well.

The Decisive Path Scheduling (DPS) algorithm [12] has been shown to be very efficient for homogeneous case. HNPD leverages the benefits of DPS on the heterogeneous processor case. To build the scheduling queue, DPS first calculates the top and bottom distance from each node, the sum of which gives the DP for each node. The top and bottom distances are calculated as per Section 2, using the mean computation value for each node. After building the DP for each node, DPS begins creating the scheduling queue in a top-down fashion starting with the DAG’s entry node and traversing down the CP (which is the largest DP from an entry node). If all of a node’s predecessors have been added to the scheduling queue, the node itself is added to the queue. If not, DPS attempts to schedule its predecessors in the same fashion. The first predecessors added to the queue are those that include the node in their DP. The time complexity of DPS is $O(v^2)$.

4. The HNPD Algorithm

4.1 Algorithm Description

HNPD is based upon DPS, which is the best algorithm for the homogeneous case. The motivation for the design of HNPD is multifold. It facilitates progress of execution along the critical path of the DAG. It assigns highest priority to critical path nodes (CPN) and then to those predecessors of CPNs that include the CPN in their DP. Unlike CPOP, it does not assign all critical path nodes to a single processor; it attempts to select the processor that satisfies the objective function best, to execute nodes in the critical path. Among these predecessors HNPD gives higher priority to nodes with higher DP values. This is because the nodes with the higher DP values are likely to be on longer paths. It uses insertion based scheduling to utilize the idle slots of processors. It also uses task duplication to suppress communication if it reduces the ready time of tasks. The duplication attempts are prioritized for nodes in decreasing order of top distance because the predecessor with the largest top distance is likely to delay the ready time of the node most. HNPD also attempts to recursively duplicate the predecessors of duplicated tasks as far as possible in a similar fashion to achieve the same objective. In the order of tasks in the scheduling queue, HNPD uses $EFT(v)$ to select the processor for each task $v$. Doing so, also exploits the heterogeneity of the systems.

HNPD is an insertion-based algorithm; therefore it calculates $T_{Available}[v, p]$ to be the earliest idle time slot large enough to execute $T(v, p)$. In other words, the algorithm looks for a possible insertion between two already-scheduled tasks on the given processor without violating precedence relationships.

Once tasks have been assigned to processors, HNPD attempts to duplicate predecessors of the tasks. Tasks are ordered from most favorite to least and by descending top distance. The goal of duplicating predecessors is to decrease the length of time for which the node is awaiting data by making use of the processor’s idle time. While execution of the duplicated predecessor may complete after its actual (in absence of duplication) $EFT$, the avoided communication cost may make the duplication worthwhile.
If there is idle time between the recently assigned task \( v_i \) and the preceding task on the processor \( p_j \), HNPD attempts to duplicate each predecessor \( v_p \). If \( v_p \) is not already scheduled on processor \( p_j \), it is duplicated if \( \text{EFT}(v_p, p_j) \) is less than \( \text{EFT}(v_i, p_j) + C(v_p, v_i) \). The duplication is retained if \( \text{EFT}(v_p, p_j) \) decreases. Otherwise, it is discarded. The same duplication procedure is repeated for each predecessor in order of most favorite to least.

### schedule_nodes

1. Compute \( DP \) for all nodes \( \mathcal{O}(v^2) \)
2. Compute \( \text{sched}_\text{queue} \) using DPS \( \mathcal{O}(v^2) \)
3. while there are unscheduled nodes in the \( \text{sched}_\text{queue} \) do \( \mathcal{O}(v) \)
   4. Select the first task \( v_i \) in the list and remove it.
   5. for each processor \( p_j \) in \( P \) \( \mathcal{O}(p) \)
   6. \( \text{max}_\text{parent}_\text{endtime} = \text{get}_\text{max}_\text{parent}_\text{time}(v_i, p_j) \)
   7. \( \text{insertion}_\text{time} = \max(\text{max}_\text{parent}_\text{endtime}, \text{EFT(last node scheduled on } p_j)) \)
   8. for each node \( v_n \) already scheduled on \( p_j \), from last to first scheduled \( \mathcal{O}(v) \)
      9. if \( v_n \) is not an ancestor of \( v_i \) then
         10. if \( v_n \) is the first node scheduled on \( p_j \) then
             11. if \( \text{EST}(v_n) \geq T(v_n, p_j) + \text{max}_\text{parent}_\text{time} \) then
                 12. \( \text{insertion}_\text{time} = \text{max}_\text{parent}_\text{time} \)
             13. else if \( [\text{EST}(v_n) - \text{EFT(node scheduled before } v_n)] \geq T(v_n, p_j) \) and
                 14. \( [\text{EFT(node scheduled before } v_n)] \geq \text{max}_\text{parent}_\text{endtime} \) then
                   15. \( \text{insertion}_\text{time} = \text{EFT(node scheduled before } v_n) \)
         16. endfor
   17. \( \text{EFT}(v_i, p_j) = \text{insertion}_\text{time} + T(v_i, p_j) \)
   18. endfor
19. Schedule \( v_i \) to the processor \( p_f \) that provides the lowest EFT (as calculated in line 17)
20. \( \text{duplicate}_\text{fpreds}(v_i, v_i, p_f) \) \( \mathcal{O}(v) \)
21. // cannot duplicate more than \( v \) nodes on each processor
22. // duplication is therefore bounded by \( \mathcal{O}(v * p) \)
23. endwhile

### get_max_parent_time(Node \( n \), Processor \( p \))

1. for each parent \( v_p \) of \( n \) \( \mathcal{O}(v) \)
2. if \( v_p \) is scheduled on \( p \) then
3. if \( \text{EFT}(v_p) > \text{max}_\text{parent}_\text{endtime} \) then \( \text{max}_\text{parent}_\text{endtime} = \text{EFT}(v_p) \) 
4. else if \( \text{EFT}(v_p) + C(v_p, n) > \text{max}_\text{parent}_\text{endtime} \) then
5. \( \text{max}_\text{parent}_\text{endtime} = \text{EFT}(v_p) + C(v_p, n) \)
6. endfor

---

**Figure 1** – HNPD Algorithm, Scheduling Routine

### duplicate_fpreds(Node \( n \), Node inserted_node, Processor \( p \))

1. \( \text{duplicate}_\text{node}(fpred(n), n, \text{inserted}_\text{node}, p) \)
2. if \( fpred(n) \) was duplicated then
3. \( \text{duplicate}_\text{fpreds}(fpred(n), \text{inserted}_\text{node}, p) \)

### duplicate_node(Node \( n \), Node succ, Node inserted_node, Processor \( p \))

1. if \( n \) is not already scheduled on \( p \) then
2. \( \text{max\_parent\_time} = \text{get\_max\_parent\_time}(n, p) \)
3. \( \text{for each node } v_n \text{ already scheduled on } p, \text{ from last to first scheduled } //[O(v)] \)
4. \( \text{if } v_n \text{ is not an ancestor of } n \text{ then} \)
5. \( \text{if } v_n \text{ is the first node scheduled on } p \text{ then} \)
6. \( \text{if EST}(v_n) >= T(n, p) + \text{max\_parent\_time} \)
7. \( \text{insertion\_time} = \text{max\_parent\_time} \)
8. \( \text{else if } [\text{EST}(v_n) - \text{EFT(node scheduled before } v_n) >= T(n, p)] \text{ and} \)
9. \( [\text{EFT(node scheduled before } v_n) >= \text{max\_parent\_endtime}] \text{ then} \)
10. \( \text{insertion\_point} = \text{EFT(node scheduled before } v_n) \)
11. \( \text{next} \)
12. \( \text{EFT}(v_i, p_j) = \text{insertion\_time} + T(v_i, p_j) \)
13. \( \text{if } \text{EFT}(n, p) < \text{EFT}(n) + C(n, \text{succ}) \text{ then} \)
14. \( \text{insert and schedule } n \text{ on } p \)
15. \( \text{recalculate } \text{EFT(} \text{inserted\_node, p} //[O(v)] \)
16. \( \text{if } \text{EFT(} \text{inserted\_node, p} \text{) improves then} \)
17. \( \text{keep } n \text{ scheduled on } p \)
18. \( \text{else} \)
19. \( \text{discard the duplication} \)

---

**Figure 2 – HNPD Algorithm, Duplication Routines**

After HNPD attempts to duplicate each predecessor, it recursively attempts to duplicate the predecessors of any duplicated tasks. Thus, as many ancestors as allowable are duplicated, in a breadth-first fashion. Duplication recursively continues until no further duplication is possible.

### 4.2 Time Complexity

The time complexity of HNPD compares well with other heuristics. HEFT has a time complexity of \(O(v^2 \times p)\) [16]. STDS has a slightly better time complexity of \(O(v^2)\) [14]. HNPD matches HEFT with a time complexity of \(O(v^2 \times p)\), as shown in Figures 1 and 2, and derived below.

DPS, used for ordering the scheduling queue, has a time complexity of \(O(v^3)\) [12]. We can also derive this complexity by considering the discovery of the DP for each node. Each node has to examine up to \(v\) nodes while building its DP, (Figure 1, `schedule_nodes`, line 1). Therefore the time complexity of finding the DP is \(O(v^3)\). When DPS builds the task queue, it must examine all edges in the DAG, so it is also bounded by \(O(v^3)\), (Figure 1, `schedule_nodes`, line 2).

When HNPD schedules each node (total of \(v\), Figure 1, `schedule_nodes`, line 3), it looks at each processor (total of \(p\), Figure 1, `schedule_nodes`, line 5) and considers the idle time between all previously scheduled tasks on the processor (maximum of \(v\), Figure 1, `schedule_nodes`, line 8). The scheduling routine therefore has a time complexity of \(O(v^2 \times p)\). Each task duplication also considers the idle time between all previously scheduled tasks for the processor (maximum of \(v\), Figure 2, `duplicate_node`, line 3). Therefore, each task duplication has a time complexity of \(O(v)\). The maximum number of duplications would be when each node is duplicated on each processor (Figure 1, `schedule_nodes`, line 20), thus also giving a time complexity of \(O(v^2 \times p)\).

### 5. Performance

From Table 1, the performance ranking of the algorithms in terms of makespan is \{HEFT, FCP, FLB\} and \{STDS, BIL\}. Therefore, we chose HEFT and STDS to compare against HNPD.
This section presents the performance comparisons of HNPD to HEFT and STDS. Each of the algorithms was run against the same randomly generated set of DAGs. Results were then correlated to produce meaningful performance metrics.

The Schedule Length Ratio (SLR) is used as the main metric for comparisons. The SLR of a DAG is defined as the makespan divided by the CPEC (as defined in Section 2). Since CPEC considers the minimum execution time for each node and ignores inter-processor communication costs, the CPEC is the best possible makespan and cannot be improved upon. Therefore, SLR can never be less than one. The algorithm that produces the lowest SLR is the best algorithm with respect to performance. The improvement percentage of HNPD over one of the heuristics is defined as the percent by which the original SLR is reduced. We can mathematically define the metrics as below:

\[
SLR = \frac{\text{makespan}}{\text{CPEC}}
\]

\[
\text{Improvement\_Percentage} = \left( \frac{\text{Comparison\_SLR} - \text{HNPD\_SLR}}{\text{Comparison\_SLR}} \right) \times 100
\]

### 5.1 Generating Random Task Graphs

The simulations were performed by creating a set of random DAGs which were input to the three heuristics. The method used to generate random DAGs is similar to that presented in [16]. The following input parameters are used to create the DAG.

- Number of nodes (tasks) in the graph, \( v \).
- Shape parameter of the graph, \( \alpha \). The height of the DAG is randomly generated from a uniform distribution with mean value equal to \( \sqrt{v} / \alpha \). The width for each level in the DAG is randomly selected from a uniform distribution with mean equal to \( \alpha \times \sqrt{v} \). If \( \alpha = 1.0 \), then the DAG will be balanced. A dense DAG (short DAG with high parallelism) can be generated by selecting \( \alpha \gg 1.0 \). Similarly, if \( \alpha \ll 1.0 \), it will generate a sparse DAG (long DAG with small degree of parallelism).
- Out degree of a node, \( \text{out\_degree} \). Each node’s out degree is randomly generated from a uniform distribution with mean value equal to \( \text{out\_degree} \).
- Communication to computation ratio, \( \text{CCR} \). If the DAG has a low CCR, it can be considered as a computation-intensive application; if CCR is high, it is a communication-intensive application.
- Average computation cost in the graph, \( \text{avg\_comp} \). Computation costs are generated randomly from a uniform distribution with mean value equal to \( \text{avg\_comp} \). Therefore, the average communication cost is calculated as \( \text{CCR} \times \text{avg\_comp} \).
- Range percentage of computation costs on processors, \( \beta \). A high \( \beta \) value causes a wide variance between a node’s computation across the processors. A very low \( \beta \) value causes a task’s computation time on all processors to be almost equal. Let \( w \) be the average computation cost of \( v_i \), selected randomly from a uniform distribution with mean value equal to \( \text{avg\_comp} \). The computation cost of \( v_i \) on any processor \( p_j \) will then be randomly selected from the range \([w \times (1 - \beta/2)] \) to \([w \times (1 + \beta/2)] \).
- Processor availability factor, \( m \). The number of available processors, \( p \), is equal to \( m \times v \). A scheduling algorithm cannot use more processors than tasks; therefore \( m \) should never be larger than one.

A set of random DAGs was generated as the study test bed. The input parameters described above were varied with the following values.
These values produce 10,800 different combinations of parameters. Since we generated 25 random DAGs for each combination, the total number of DAGs in the study is around 270,000.

5.2 Local Optimization

We investigated duplication of predecessors based upon local CCR values. The local CCR was calculated by dividing the communication cost between the predecessor and the inserted node by the computation cost of the predecessor on the processor under consideration. If this value was greater than or equal to one, duplication was attempted. The heuristic performed worse than when duplicating all predecessors, regardless of the local CCR value. For CCR values of 0.1, the performance improved by less than 1%; for CCR values of 0.5 and 1.0, the performance decreased by 3%; for CCR values of 5.0 and 10.0, the performance was the same. Therefore, we determined that it is more productive to attempt to duplicate all predecessors regardless of local CCR.

5.3 Results

The performances of HEFT, STDS and HNPD were compared using the average SLR for different test sets of random DAGs. The test sets are created by combining results from DAGs with similar properties, such as the same number of nodes or the same CCR. The results below show performance when attempting to duplicate all predecessors. In this section, we will present the test set combinations that provide the most meaningful insight into performance variances.

The first test set is achieved by combining DAGs with respect to number of nodes. The SLR value was averaged from 54,000 different DAGs with varying CCR, α, β, out_degree and m values, and avg_comp equal to 100. Figure 3 plots the average SLR value with respect to number of nodes. The performance ranking is {HNPD, HEFT, STDS}. HNPD performs slightly better than HEFT at all points. The percentage improvement increases as the number of nodes increases, with HNPD outperforming HEFT by 6.4% for 10 nodes to around 20% for 1000 nodes. HNPD performs much better than STDS, with the performance difference widening as the DAG size increases. HNPD performance is even with STDS for 10 nodes, 12% for 100 nodes and about 3% for 1000 nodes. This is because STDS benefits with increase in number of nodes, as it can use unlimited number of processors. For all graph sizes the average SLR of HNPD is less than 3 (a rough empirical bound) and for very small and very large graphs average SLR of HNPD is less than 2.
Figure 3. Average SLR with respect to Number of Nodes

The second test set combines DAGs with respect to CCR. Each SLR value for this set is averaged from 54,000 different DAGs with varying v, α, β, out_degree and m values, with avg_comp again equal to 100. In Figure 4, average SLR is plotted with respect to varying CCR. Again, the performance ranking is {HNPD, HEFT, STDS}. For CCR values less than or equal to one (i.e. DAGs that are computation-intensive), HNPD and HEFT perform almost exactly even. However, as CCR becomes larger than one (i.e. DAGs that are communication-intensive), HNPD clearly outperforms HEFT. For CCR values of 0.1, HEFT actually outperforms HNPD by 2.5%. For CCR value of 0.5, the difference is less than 0.5%. For CCR value of 1.0, HNPD outperforms HEFT by 4%. Then, the true savings begin to be achieved; for CCR values of 5.0 and 10.0, HNPD realizes savings of around 21% and 27%, respectively. HNPD outperforms STDS at each point with the performance difference largest for CCR values less than one. HNPD outperforms STDS by 19.8% for CCR equal to 0.1 down to 7% for CCR equal to 5.0. The savings returns to 14% for CCR equal to 10.0. This trend is the opposite of that displayed with HEFT.

Figure 4. Average SLR with respect to CCR
The third test set combines DAGs with respect to $m$. For this test set, each SLR value is averaged from 90,000 randomly generated DAGs with varying $v$, $CCR$, $\alpha$, $\beta$, $out\_degree$, and a static $avg\_comp$ of 100. Figure 5 shows the graph of average SLR to varying $m$. As before, the performance ranking is {HNPD, HEFT, STDS}. HNPD outperforms HEFT at each point, with the performance difference slightly increasing as $m$ increases. For $m$ equal to 0.25, 0.5 and 1.0, HNPD outperforms HEFT by 18%, 20.4% and 20.6%, respectively. HNPD also outperforms STDS at each point, with the performance improvement being drastically greater for low values of $m$. As the number of processors approaches the number of nodes (i.e., $m$ approaches 1.0), STDS begins to perform more efficiently. Yet, HNPD still outperforms STDS even for $m = 1.0$. The performance difference for $m$ equal to 0.25, 0.5 and 1.0 is 27.6%, 18.8% and 14.2% respectively. Again, this trend is the exact opposite of the trend seen with HEFT.

![Figure 5. Average SLR with respect to Processor Availability Factor](image)

### 5.4 Performance Results for Unlimited Processors

By scheduling DAGs on unlimited number of processors, more insight may be gained about the relative performance of STDS vs. HNPD. Since STDS duplicates tasks by creating new clusters on new processors, STDS needs many processors to perform well and works best with an unbounded number of processors. By comparing HNPD to STDS for the unlimited processor scenario, we can see how HNPD performs under best conditions for STDS. We can assume that employing the number of processors is equal to the number of nodes (i.e. $m = 1.0$) can mimic unlimited processors. A scheduler should never employ more processors than the number of nodes. In the results below, we have shown the percentage improvement of HNPD compared to both HEFT and STDS.

In Figure 6, the result from our unlimited processor test shows the comparison of $CCR$ to the improvement percentage. This test was run for random DAGs with $m$ equal to 1.0, $avg\_comp$ equal to 100, and all other variables varying. A total of 90,000 DAGs were generated. Holding consistent with the overall results above, HNPD outperforms HEFT at every point, with performance improving as $CCR$ increases. However, the results from this test run reveal that STDS performs close to HNPD for DAGs with large $CCR$ values and unlimited processors. Since STDS performs duplication by creating new clusters and utilizing more processors, it has the flexibility to perform task duplication in cases where HNPD cannot. In order to duplicate a node, HNPD requires idle time on a specific processor, whereas STDS simply requires a free
processor. In the unlimited processor scenario, STDS is allowed to perform unlimited duplication. Therefore, these savings are not seen when using STDS with a bounded set of processors.

![Figure 6. Improvement by CCR, Unlimited Processors (m = 1.0)](image)

**5.5 Performance on Few Processors**

Similarly, we can gain more insight into the comparison of HEFT and HNPD by looking at the opposite extreme, a small set of processors. HEFT was designed to work with a bounded set of processors. As Figure 5 showed, HEFT performed closest to HNPD when \( m \) was equal to 0.25, or when there were only one-fourth as many processors as nodes in the system.

The result from our test with a small set of processors, shown in Figure 7, compares percentage improvement with respect to CCR. Percentage Improvement is defined the same as in our unlimited processor test. The random DAGs for this test were created with \( m \) equal to 0.25, \( \text{avg}_{\text{comp}} \) equal to 100, and all other variables varying. The test set included 90,000 DAGs.

For the small set of processors, STDS does not perform well at all, and HNPD produces huge savings. HEFT, however, outperforms HNPD for the small set where CCR is less than or equal to one, as shown in Figure 7. Again, we point out that HEFT greatly outperforms STDS for the same set of DAGs and that performance improvement of HNPD over HEFT for high CCR values balances HEFT’s advantage in calculating DAGs with a low CCR.

HEFT outperforms HNPD for low CCR values and a small set of processors for the following reasons. Low CCR values indicate a computation-intensive application. In these cases, task duplication may not provide much relief in reducing the makespan, since task duplication only eliminates communication cost. Indeed, HNPD may actually hinder later task scheduling by filling the processor idle time with duplicated predecessors. While the duplication may at the time provide a small saving to the makespan, the processor will be unavailable later for future tasks. These tasks must then be executed later or on another, less efficient processor. Since HEFT does no task duplication, HEFT uses the idle time to schedule other tasks.
5.6 Performance by $\alpha$, $\beta$, out_degree and $m$

HNPD performs better for all shape parameters over HEFT and STDS with performance improving substantially over STDS as the graph becomes wider. This is because HNPD systematically considers all predecessors for duplication and prioritizes predecessors of CPN that include CPN in their DP.
Figure 9. Improvement by Heterogeneity

HNPD performs better for all heterogeneity over HEFT and STDS with performance improving substantially over STDS with increasing heterogeneity.

Figure 10. Improvement by Out-degree

HNPD performs better for all average out-degrees over HEFT and STDS with performance improving substantially over STDS for average out-degree more than 2.

Figure 11. Improvement by number of processors employed
HNPD uses slightly (<5%) more number of processors than HEFT to get the same makespan as it performs task duplication, but 40% fewer processors than STDS to get the same makespan, although both perform task duplication.

### 5.7 Performance Conclusions

Interestingly enough, HNPD is outperformed only at the two extremes. STDS performs close to HNPD (and outperforms HEFT) for unlimited processors and high CCR values. HEFT outperforms HNPD (and STDS) for a small set of processors and low CCR values. However, these scenarios are small enough that HNPD still performs better than STDS overall for an unlimited number of processors, and better than HEFT overall for a small set of processors.

### 6. Conclusions

In this paper we have proposed a new task-scheduling algorithm, Heterogeneous N-Predecessor Duplication (HNPD), for scheduling parallel applications on heterogeneous processors. By running a suite of tests with randomly generated DAGs, we showed that HNPD outperforms both HEFT and STDS in terms of scheduling efficiency. HNPD has been shown to be very consistent in its scheduling results, performing well with both an unlimited number of processors and a small set of processors. The performance improvement is across all graph shapes and out-degrees and processor heterogeneity. Further research may be done to extend HNPD so that it performs even better in the extreme cases, such as high CCR values on unlimited processors and low CCR values for the small set of processors.

### References


