Communication Aware Multiprocessor Binding for Shared Memory Systems

Shreya Adyanthaya[1], Marc Geilen[1], Twan Basten[1,2], Jeroen Voeten[2,1], Ramon Schiffelers[3,1]
[1]Eindhoven University of Technology, Eindhoven, The Netherlands
[2]TNO Embedded Systems Innovation, Eindhoven, The Netherlands
[3]ASML, Veldhoven, The Netherlands
{s.adyanthaya, m.c.w.geilen, a.a.basten, j.p.m.voeten}@tue.nl, ramon.schiffelers@asml.com

***Abstract*—We present a three-step binding algorithm for applications in the form of directed acyclic graphs (DAGs) of tasks with deadlines, that need to be bound to a shared memory multiprocessor platform. The aim of the algorithm is to obtain a good binding that results in low makespans of the schedules of the DAGs. It first clusters tasks assuming unlimited resources using a deadline-aware shared memory extension of the existing dominant sequence clustering algorithm. Second, the clusters produced are merged based on communication dependencies to fit into the number of available platform resources. As a final step, the clusters are allocated to the available resources by balancing the workload. The approach is compared to the state of the art bounded dominant sequence clustering (BDSC) algorithm that also performs clustering on a limited number of resources. We show that our three-step algorithm makes better use of the shared memory communication structure and produces significantly lower makespans than BDSC on benchmark cases.**

I. Introduction

To meet the growing demands of embedded system applications, there is a shift from single-core to multi-core execution platforms. Utilizing the parallelism in the applications and executing tasks simultaneously on parallel cores can result in significant improvement in performance. However, exploiting parallelism may induce significant communication overhead when tasks run on different resources. Additionally, the size of industrial applications is typically large and the available resources are limited. This calls for a binding algorithm that, taking platform size constraints into account, utilizes the application computation and communication structure to produce a good binding of tasks on the platform resources.

We target real time applications modeled as directed acyclic graphs (DAGs) that are to be bound to homogenous shared memory multiprocessor platforms. Tasks in these graphs have execution times and deadlines. Tasks are not preempted. Shared memory communication is a commonly used paradigm for multi-core systems wherein multiple processors access the same shared memory. Tasks running on these processors communicate with each other via this shared memory. The motivation for this communication paradigm also comes from our case study of the industrial binding challenges of ASML, the world's leading provider of lithography machines called wafer scanners [1]. These machines include large control applications that need to be bound to the cores of multiprocessors that share memory. Our specific aim is to apply our method to large applications such as those of ASML. The exact configuration parameters that are needed to complete the applications are set during machine initialization. As such, the binding and scheduling algorithms run during machine start-up time and need to be fast. In this work, we distinguish binding and scheduling as separate concerns. The motivation behind this is to narrow down the scope of the problem to binding and focus on obtaining a good binding method that optimizes the makespans of the schedules produced using that binding while ensuring that task deadlines are met. Under the assumption that read-write operations are part of the task execution, shared memory communication mainly includes synchronization operations. Depending on the schedule order, many of these operations may be redundant due to the ordering being indirectly enforced by other synchronizations. For instance, a receiver task need not synchronize with a sender task if another receiver task scheduled before it on its resource has already synchronized with that particular sender. Hence, the exact synchronization needed is only known after the schedule is formed and not during binding. This lack of complete knowledge makes the binding problem challenging for shared memory systems.

The main contribution of this work is a binding approach for shared memory systems that includes three steps, namely clustering, merging and load balanced allocation. We refer to our binding algorithm as CMA ('C'lustering, 'M'erging, 'A'llocation). The clustering step uses a deadline-aware shared memory extension of the Dominant Sequence Clustering (DSC) algorithm [2] to produce clusters of tasks in the graph without considering platform constraints. Since the number of clusters thus formed can be higher than the number of platform resources, the next step merges clusters using the application structure while constraining the merging based on platform size. The final step then allocates clusters to resources balancing their load. After the binding, a scheduler will order the tasks on the processors. We validate CMA by drawing comparisons with the state of the art bounded dominant sequence clustering (BDSC) algorithm [3] which is an extension of DSC for limited resources. We compare them on a benchmark consisting of industrial applications of ASML wafer scanners and a large number of generated test cases of well known parallel applications. We show that our algorithm outperforms BDSC in most of the cases.

The paper is organised as follows. Section II gives preliminaries and Section III gives the problem definition and solution flow. Section IV explains the first clustering step. Section V and VI elaborate on the merging and resource allocation steps. Section VII illustrates our experimental evaluation. Section VIII discusses related work. Section IX concludes.

II. Preliminaries

We denote the sets of real numbers with $\mathbb{R}$, non-negative real numbers with $\mathbb{R}^{\geq 0}$ and integers with $\mathbb{Z}$. We use X^* to denote the set of lists with elements from set X.

An *application* is a DAG, $G = (T, D)$ with a finite set T of tasks and the set of dependencies between tasks $D \subseteq$

$(T^2 \times \mathbb{R})$. A *multiprocessor platform* is a set of processors called *resources* denoted by R.

A *task* $t \in T$ is defined by a tuple $t = (e_t, r_t, d_t) \in \mathbb{R}^{\geq 0} \times (R \cup \{\perp\}) \times \mathbb{R}^{\geq 0}$, where e_t denotes the execution time, r_t denotes the resource and d_t denotes the deadline of t. The value $\perp$ denotes that the task is not (yet) bound to a resource. Initially r_t is $\perp$ and the binding algorithm must select and bind t to one of the resources in R. A *dependency* $(a, b, \hat{c}) \in D$ implies that it takes $\hat{c}$ time units for b to obtain the data communicated from a only after which it can begin execution. Here, a, b are the source and destination tasks; $\hat{c}$ is the communication cost.

A *(static-order) schedule* S is a mapping $S : R \rightarrow T^*$ that maps a resource to an ordered list of tasks in which all data dependencies are respected. No task begins its execution before the completion of any task having a direct or indirect dependency to it. At runtime, tasks execute in the order given by the schedule as soon as their respective resource is available and their dependencies are satisfied. We let s_t denote the start time and c_t denote the completion time of task t. The completion times are obtained by adding task execution times to task start times implying that $c_t = s_t + e_t$. The completion time of the task completing last is the *makespan* m of the schedule and is given by $m = \max_{t \in T} c_t$.

III. Problem definition and solution flow

A. Problem definition

There are two main kinds of communication paradigms seen in practice, namely message passing systems and shared memory systems. In message passing systems tasks communicate by sending messages through a communication network and communication costs are incurred on dedicated communication resources. Our work is focused on shared memory systems where tasks communicate by reading and writing variables in a shared memory infrastructure. This happens when the tasks are bound to cores of a multi-core processor that share memory. Since shared memory read-write operations are typically non-blocking, we must ensure that the receiver task reads data from the memory only after the sender task has written into the memory. To ensure that the data flow semantics are followed, shared memory systems can make use of data synchronization mechanisms. The mechanism considered in this paper concerns updating a status flag by the sender, indicating that it has completed writing its data. The receiver task, once enabled, waits until the status flag is set to read the data. In our work, the time taken to read and write data to the shared memory is accounted for in the execution time of the task. We make the approximation of assuming that there is no contention on the shared memory. Communication cost is thus composed of (1) the time taken by the sender task to update the status flag (U), and (2) the time taken by the receiver task to read the status update (R). The receiver task has to one by one synchronize with each sender task from which it has an incoming dependency. These communication costs are induced by communication (or synchronization) operations that are added in the schedule and executed on the same resources as the sender and receiver tasks respectively. There is no dedicated communication network and the communication costs are incurred by the computation resources. In this work, we assume communication costs to be fixed for a graph and ignore varying timings that can be caused by cache operations. Figure 1(a) is an example of shared memory communication, where the dotted lines show the dependencies in the graph.

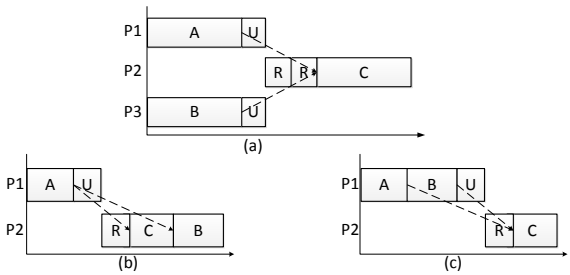

Fig. 1. (a) Example of communication in shared memory systems, (b) No read needed for B due to transitive reduction, (c) No read needed for C from A and correspondingly no update needed for A

Task C synchronizes with tasks A and B. There is a read operation per predecessor for C on its processor. Although a task needs a read operation per predecessor, it updates its status flag only once irrespective of its number of successors. Hence, the number of reads is at least equal to and can often be more than the number of updates.

The synchronization operations in schedules of shared memory systems may sometimes be redundant depending upon whether the synchronization has implicitly taken place due to synchronizations with other tasks scheduled earlier. For instance consider Figure 1(b) where a task B is waiting to synchronize with a predecessor A. This synchronization is redundant if another task C scheduled before B on its processor has already synchronized with A. In this case, the data from A is already in the memory and its flag has been updated. Then the synchronization for B can be removed. This process of removing redundant communications that are implied by others is known as transitive reduction [4]. If all read operations corresponding to an update have been transitively removed, the update is also removed. This is shown in Figure 1(c) where the synchronization for the dependency between A and C is implied by that between B and C. The read operation for C from A is removed after which the update from A is not needed and is also removed. To remove such redundant synchronization, the number of communication operations needs to be optimized by performing transitive reduction on the graphs after scheduling. This is because the reduction depends on the binding and scheduling order of the tasks. For instance, in Figure 1(b) if B is scheduled before C, then the read for C is redundant instead of the read for B. Alternatively, if B is bound to a third resource $P3$, then no transitive reduction is possible. Since this binding and ordering is unknown during the binding phase, it is not possible to predict the exact amount of synchronization that will appear in the final schedule. As binding algorithms typically exploit this information, this makes the binding problem challenging. The problem statement is given below.

Problem Statement. *Given an application and a shared memory multiprocessor platform, find a binding of the application tasks on the available platform resources that, upon scheduling, gives the lowest makespan with all task deadlines being met.*

As this problem is NP-complete, in analogy with the partitioning problem in graph theory [5], finding an optimal solution is not feasible. So we present an algorithm that uses heuristics to produce a good binding.

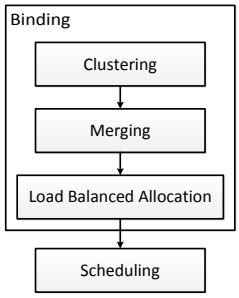

Fig. 2. Solution Flow

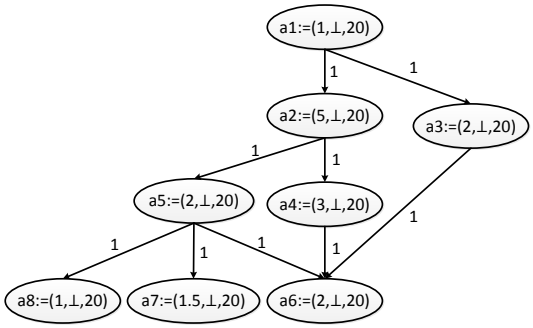

Fig. 3. An example DAG with the binding unknown

B. Solution flow and rationale

Our solution involves three steps, namely clustering of tasks constraining them to be bound to the same resource, merging and load balanced allocation as shown in the flow chart in Figure 2. The benefit of having a three-step approach instead of a single step one is the use of more high level (global) structure of the graph in comparison to a single step approach like BDSC thats makes local binding decisions. After the first clustering step on unlimited resources we have the global information of all the cluster dependencies. The merging step bases its decisions on this global knowledge. BDSC, on the other hand, makes local decisions by assigning a task to one of the available clusters with the best timings for the task once the number of clusters has reached the resource limit.

The use of global information is particularly beneficial for shared memory systems because transitive reduction cannot be performed during clustering and we do not know accurate communication costs during binding. Local decisions of BDSC made based on inaccurate timing values may not be good. We attempt to reduce the impact of this inaccuracy by first making as many clusters as possible assuming unlimited resources and then merging highly connected clusters together. The hypothesis is that there is a higher likelihood of highly connected clusters having higher communication overhead between them even after transitive reduction in comparison to less connected clusters. Hence, they should be merged. After merging, clusters are allocated to resources by balancing the workload. The end result is a graph with known binding which is then fed into a scheduler to decide the task orderings.

IV. Clustering

In this section we present our solution for clustering the tasks in the graph using a deadline-aware modified DSC algorithm that can deal with shared memory type of communication. We first briefly explain DSC followed by details of the extension and the algorithm itself.

A. DSC

DSC follows a list scheduling [6] pattern where at each step a new task is clustered of which all predecessors have been clustered before. We refer to such a task as an enabled task. From the enabled tasks, DSC selects a task on the longest path from source to sink in a partially clustered DAG. This path is called the dominant sequence. For a fully clustered DAG, this is the critical path. To detect which task among the enabled tasks is on the dominant sequence, DSC assigns priorities to tasks equal to the length of the longest path that passes through them. The objective of this selection is to minimize the length of the dominant sequence by avoiding communication costs along this path first. This is achieved by assigning tasks to the same clusters which in turn is expected to reduce the length of the critical path and thus the makespan.

Definition 1. (Cluster) *A cluster $cl \subseteq T$ is a set of tasks that are constrained to be bound to the same resource. The size $sz(cl)$ of cl is the sum of the execution times of its constituent tasks. There exists a cluster dependency (cl, cl') if there is a dependency $(a, b, \hat{c}) \in D$ between some tasks $a \in cl$ and $b \in cl'$. We use CLD to denote the set of cluster dependencies.*

Since tasks in a cluster are assigned to the same resource, the communication costs between them is zero. The length of the longest path that passes through a task is calculated as the sum of the task's top level and bottom level, defined below.

Definition 2. (Top Level) *The top level of a task t in DAG G, denoted by $tlevel(t, G) \in \mathbb{R}$, is the length of the longest path from any of the source nodes (tasks without incoming dependencies) of G to t. The length of a path is composed of the computation costs of the tasks and the communication costs of the edges along the path.*

Definition 3. (Bottom Level) *The bottom level of a task t in DAG G, denoted by $blevel(t, G) \in \mathbb{R}$, is the length of the longest path from t to any of the sink nodes (tasks without outgoing dependencies) in G.*

Consider the example DAG in Figure 3 which needs to be bound on a platform consisting of 2 processors $P1$ and $P2$. The numbers shown next to the dependencies between the tasks specify the communication cost. The cost of the read and update operations is taken to be 0.5 units each giving a total communication time of 1 time unit. The initial top levels and bottom levels of tasks as computed by DSC are given in Table I. The last column will be explained later. The task having the highest $tlevel + blevel$ among the enabled tasks is on the dominant sequence and is chosen to be clustered next. Priorities are assigned as $tlevel + blevel$ as this gives the length of the dominant sequence and thus has more information than using, for instance, $blevel$ alone which only gives the length of the path from a current node to an exit node. In the example, the dominant sequence is a_1, a_2, a_4, a_6.

The task that is selected to be clustered is added to the cluster that gives it the highest reduction in $tlevel$ resulting from zeroing the communication cost of edges from predecessors in the cluster, if such a cluster exists. In computing the new reduced $tlevel$ on the cluster, the DAG is also updated by adding a dependency from the last task in the cluster to the selected task. This is to account for the tasks that were already added to the cluster before the current selected task. These additional dependencies in the graph are temporary and are

TABLE I
INITIAL DSC TOP-LEVELS, INITIAL DSC BOTTOM-LEVELS AND DUE-DATES OF TASKS IN THE EXAMPLE DAG

Task	*tlevel*	*blevel*	*dd*
a_1	0	14	10
a_2	2	12	15
a_3	2	5	18
a_4	8	6	18
a_5	8	5	17.75
a_6	12	2	20
a_7	11	1.5	20
a_8	11	1	20

used only within the clustering phase for $tlevel$ computations. If there is no such cluster that reduces the $tlevel$, the task is assigned to a new cluster and its $tlevel$ remains the same. After each clustering step, the top levels of un-clustered tasks must be recomputed taking into account the changes in the $tlevel$ of their clustered predecessors. The complexity of performing this re-computation per step can be reduced by incrementally computing and updating top levels of enabled tasks.

B. Deadline-aware extension to DSC

In our work, we consider DAGs of tasks with deadlines. These deadlines must be taken into account for task priorities to avoid clustering decisions to cause deadline misses. In our extension to DSC, we compute priorities based on top levels and due-dates [7], defined below.

Definition 4. (DUE-DATE) *A due-date of a task t, denoted by $dd_t \in \mathbb{R}$, is an upper bound on the completion time of the task which, if missed, renders the schedule infeasible. If a task misses its due-date, at least one future task (including itself) in the schedule will miss its deadline.*

The due-date of a task t depends on its own deadline and the due-dates of its immediate successors. If t has no successors, its due-date is its deadline. The due-date of t is computed from the due-dates of its successors $succ(t)$ using Equation 1.

$$dd_t = \min\left\{ d_t, \min_{s \in succ(t)}(dd_s - e_s), \left(\max_{s \in succ(t)} dd_s - \frac{\sum_{s \in succ(t)} e_s}{|R|} \right) \right\} \quad (1)$$

This equation is composed of three terms: 1) The due-date of a task is at most its deadline d_t. 2) Each successor of t has to complete before its due-date. Hence, t must finish its execution before the minimum of $(dd_s - e_s)$ over all the successors. 3) The third term exploits the fact that all successors must complete before the maximum of their due-dates. For example in Figure 3, all successors of a_5 must be completed before the maximum of their due-dates equal to 20. The execution time for the total workload cannot be less than the averaged workload over the set of resources. Hence, t must be completed before the average successor workload deducted from the maximum of the successor due-dates. The averaged workload of the successors of a_5 over the set of resources $\{P1, P2\}$ is $4.5/2 = 2.25$. Hence, the value of this term for a_5 is $20 - 2.25 = 17.75$. Due-dates of all tasks are computed before the start of the scheduling process, by a single backward traversal through the graph. For Figure 3, starting from leaf nodes a_6, a_7 and a_8 and having their deadlines as due-dates, the remaining task due-dates are computed using Equation 1 in the last column of Table I.

Since tasks are not yet scheduled during the computation of due-dates, it is not possible to have a good estimate of the amount of communication penalty that needs to be taken into account during the computation. Hence, we compute a conservative estimate of the due-dates by not taking any communication overhead into account. Given the task due-dates and top levels, we assign priorities based on the slack sl between their due-dates and their top levels increased with their execution times. The smaller the slack, the higher the priority.

$$sl_t = dd_t - (tlevel(t, G) + e_t) \quad (2)$$

Algorithm 1: Clustering()

Input : $G := (T, D)$
Output: CL, CLD

1 $CT := \emptyset$, $CL := \emptyset$, $CLD := \emptyset$;
2 **while** $CT \neq T$ **do**
3 $\quad ET := \{t \in T \mid pred(t) \subseteq CT\}$;
4 $\quad \forall t \in ET, tlevel(t, G) :=$ getTLevel$(t, \emptyset)$;
5 $\quad t :=$ arg $\min_{t \in ET} dd_t - (tlevel(t, G) + e_t)$;
6 $\quad cl_t := \{cl \in CL \mid$getTLevel$(t, cl) < tlevel(t, G)\}$;
7 $\quad$ **if** $cl_t = \emptyset$ **then**
8 $\quad\quad cl_t :=$ createNewCluster();
9 $\quad\quad CL := CL \cup \{cl_t\}$;
10 $\quad$ **end**
11 $\quad tlevel(t, G) :=$ getTLevel(t, cl_t);
12 $\quad$ **for** $t' \in pred(t)$ **do**
13 $\quad\quad$ **if** $cl_{t'} \neq cl_t$ **then**
14 $\quad\quad\quad CLD := CLD \cup \{(cl_{t'}, cl_t)\}$
15 $\quad\quad$ **end**
16 $\quad$ **end**
17 $\quad CT := CT \cup \{t\}$
18 **end**
19 **return** CL, CLD;

The clustering algorithm is given in Algorithm 1. The due-dates of all tasks are computed before clustering by one backward traversal through the graph and using Equation 1. As a pre-processing step, we perform transitive reduction on the DAG thus removing dependencies that are already redundant in the graph. CL represents the set of formed clusters, ET represents the set of enabled tasks and the set CT holds all clustered tasks. In the beginning the sets CL and CT are empty. We refer to set of all predecessors of a task t as $pred(t)$. We compute the top levels of all tasks from the list of enabled tasks in line 4. The highest priority enabled task is chosen in line 5. Line 6 assigns it to the cluster that gives the lowest updated top level that is below its current top level. If there is no such cluster then a new cluster is created with the task in lines 7-10. Line 11 updates the $tlevel$ of the task to the one on its chosen cluster. Lines 12-16 add a cluster dependency for every dependency between the chosen task and its predecessors on other clusters. In line 17, the task is added to the set of clustered tasks. This process repeats until all tasks have been clustered. The algorithm returns the set CL of clusters and the set CLD of cluster dependencies. Its complexity is $O((|T| + |D|)log|T|)$, which is the same as the complexity of DSC [2].

C. Shared memory extension to DSC

In this section, we adapt the $tlevel$ computation in the clustering algorithm of DSC, which was designed for message

Algorithm 2: getTLevel()

Input : t, cl
Output: $tlevel(t, G)$

1 **if** $cl \neq \emptyset$ **then**
2 | t_{last} := last task clustered in cl;
3 | **if** $t_{last} \neq \emptyset$ **then**
4 | | $D = D \cup \{(t_{last}, t, 0)\}$;
5 | **end**
6 **end**
7 $tlevel(t, G) := \max_{p \in pred(t)} (tlevel(p, G) + e_p)$;
8 **if** $cl = \emptyset$ **then**
9 | $tlevel(t, G) := tlevel(t, G) + commCost * |pred(t)|$;
10 **end**
11 **else**
12 | **for** $p \in pred(t)$ **do**
13 | | **if** $cl_p \neq cl$ **then**
14 | | | $tlevel(t, G) := tlevel(t, G) + commCost$;
15 | | **end**
16 | **end**
17 **end**
18 **return** $tlevel(t, G)$;

passing behaviour, to deal with shared memory systems. Communication operations are scheduled on computation resources in shared memory systems. As no transitive reduction can be performed during the top level computation, we need to make useful approximations about the synchronization costs to make better timing predictions. Firstly, we only consider read operations and ignore the update operations in the top level computations, because there is at most one update operation per sender as opposed to multiple possible read operations per receiver for all its predecessors. Hence, the read operations form the primary varying factor in the communication costs. Due to the lower impact of the update operations and the likelihood that the update operation will be removed later during scheduling and transitive reduction, we make the approximation of not considering them at all. Algorithm 2 gives the computation of the top level of t for shared memory systems. If no cluster is given as input, the algorithm returns the top level before assigning it to a cluster. If a cluster is given as input, the algorithm returns the top level in the cluster. In this case, it adds a dependency from the last task in the cluster to t in lines 1-6. Line 7 computes the maximum of the sum of top level and execution times of the predecessors of t. The top level is obtained by adding the total communication cost to this value. When no cluster is specified, then the total communication cost is the sum of the communication costs from all predecessors in lines 8-10, where $commCost$ is the cost of one read operation. When a cluster is specified, the total communication cost is the sum of the communication costs for only the predecessors in other clusters in lines 11-17. We take the sum of the communication costs (as opposed to maximum in DSC) because the synchronization operations are executed on the same resource as the task.

Table II shows the process of the clustering algorithm on the DAG in Figure 3. The first column gives the clustering step. The task chosen in that step, its top level before being assigned to a cluster, its due-date and its slack are given in the second to fifth columns. The remaining columns give the top levels in the available clusters with the * indicating the cluster

TABLE II
CLUSTERING ALGORITHM ON THE EXAMPLE DAG

Step	Task	*tlevel*	*dd*	*sl*	Top levels on clusters			
					cl_1	cl_2	cl_3	cl_4
1	a_1	0	10	9	0*			
2	a_2	1.5	15	8.5	1*			
3	a_4	6.5	18	8.5	6*			
4	a_5	6.5	17.75	9.25	9	6.5*		
5	a_7	9	20	9.5	9.5	8.5*		
6	a_8	9	20	10	9.5	10	9*	
7	a_3	1.5	18	14.5	9	10.5	10.5	1.5*
8	a_6	10.5	20	7.5	10*	11	11.5	10

TABLE III
BDSC ON THE EXAMPLE DAG

Step	Task	*tlevel*	*blevel*	*DS*	Top level on clusters	
					cl_1	cl_2
1	a_1	0	14	14	0*	
2	a_2	2	12	14	1*	
3	a_4	8	6	14	6*	
4	a_5	8	5	13	9	7*
5	a_7	11	1.5	12.5	10	9*
6	a_8	11	1	12	10*	10.5
7	a_3	2	5	7	11	10.5*
8	a_6	12	2	14	13.5	12.5*

that is chosen. In the first step only task a_1 is enabled and is added to a new cluster cl_1. After a_1 is clustered, a_2 and a_3 become enabled. As a_2 has a lower slack it is clustered on cl_1 which gives a reduction in its top level. Notice that the top level of a_2 is 1.5 as we have only considered the read part of the communication which is 0.5 time units. Next, a_4 is clustered on cl_1 after which a_5 is added to a new cluster cl_2 since cl_1 increases its top level. Then, a_7 is added to cl_2 and a_8 is added to a new cluster cl_3 as cl_1 and cl_2 increase its top level. The next task a_3 has a much smaller top level than on both cl_1 and cl_2 and is added to a new cluster cl_4. Although a_3 has been enabled since step 2, it was not chosen until step 6 due to its low priority resulting from its high slack. Finally, a_6 is added to cl_1. In Section V, we will merge clusters to reduce them to the number of resources.

D. BDSC

BDSC is a one-step approach to bind DAGs on a limited number of resources [3]. It uses an additional heuristic that checks for and fills up the idle slots at the end of existing clusters to limit the number of new clusters created. Once the number of clusters reaches the resource limit, it assigns tasks to one of the available clusters that gives it the lowest top level. In [3], it is shown to outperform other binding methods for limited resources and is therefore our benchmark for comparisons. Table III gives the outcome of applying BDSC to the DAG from Figure 3. The top level computations use the full communication cost of 1 time unit here. The clustering until step 5 is the same as in our algorithm. In step 6, task a_8 chooses cl_1 since BDSC disallows new clusters to be created after the resource limit is reached. Next, a_3 and a_6 are clustered into cl_2 finally resulting in tasks a_1, a_2, a_4, a_8 being bound to one processor and a_3, a_5, a_6, a_7 to the other.

V. MERGING

Algorithm 1 produces clusters assuming unlimited resources in the platform. The merging step combines highly connected clusters to bring down the number of clusters towards the number of available resources. The merging is based on the number of cluster dependencies. Highly mutually connected

Algorithm 3: Merging()

Input : CL
Output: CL_{merged}

1 **if** $|CL| \leq |R|$ **then**
2 **return** CL;
3 **end**
4 **else**
5 CL_{Sorted} = Clusters sorted in descending order of number of cluster dependencies with their $HCCs$;
6 **for** $cl \in CL_{Sorted}$ **do**
7 **if** $HCC(cl) = null$ **then**
8 **return** CL_{Sorted}
9 **end**
10 **if** $sz(cl) + sz(HCC(cl)) \leq threshold$ **then**
11 $CL_{Sorted} = CL_{Sorted} \cup \{(cl + HCC(cl))\}$;
12 Update CLD, HCC, CL_{Sorted};
13 $CL_{Sorted} = CL_{Sorted} \backslash \{cl, HCC(cl)\}$;
14 Update CLD, HCC;
15 **end**
16 **if** $|CL_{Sorted}| = |R|$ **then**
17 **return** CL_{Sorted};
18 **end**
19 **end**
20 **return** CL_{Sorted};
21 **end**

clusters are merged to save the communication costs between resources after the binding.

Definition 5. (HIGHEST CONNECTED CLUSTER) *A highest connected cluster function maps a cluster cl with one of the clusters it has the highest number of cluster dependencies with. It is denoted by $HCC : CL \rightarrow CL$. If the cluster has no dependencies to any other cluster, it returns null.*

The merging algorithm is given in Algorithm 3. It takes the set CL of clusters and returns the set CL_{merged} of merged clusters. If the number of clusters is less than or equal to the number of resources, we stop the process in lines 1-3. In line 5, the clusters are sorted in descending order of the number of cluster dependencies they have with their highest connected clusters. Counting the number of cluster dependencies instead of the costs of these dependencies is sufficient here because the communication costs are proportional to the number of dependencies due to the costs being fixed. The algorithm then chooses the highest connected cluster from the sorted list. If this cluster has no highest connected cluster, we have only clusters left with no cluster dependencies. The process then stops and returns the clusters. Note that there can still be more clusters than the number of resources. Otherwise it merges the chosen cluster with its highest connected cluster if the sum of the two clusters does not exceed a *threshold* given by the total workload of the task graph divided by the number of resources.

$$threshold := \frac{\sum_{t \in T} e_t}{|R|} \tag{3}$$

This enforces that the combined size of the merged clusters does not exceed the average workload of the graph on the given resource set. The motivation for this threshold is to ensure that too many clusters are not merged together resulting in unevenly sized clusters whose load cannot be evenly balanced on the resources thus adversely affecting the makespan. If the combined size is within the threshold, the chosen cluster is merged with its highest connected cluster while updating CLD, HCC and the sorted cluster list accordingly in lines 10-15. If the number of resulting clusters are equal to the number of resources, the clusters are returned in lines 16-18. Otherwise, the process repeats until the end of the list and returns the clusters. Since we sort the clusters, each time two clusters are merged the sorted list along with the relevant cluster dependencies need to be updated to include the new cluster and remove the constituent ones. Hence, the worst case complexity of this algorithm is $O(|T|(|T| + |D|))$. In the example of Figure 3, the average workload is 17.5/2 = 8.75 time units. On top of the sorted cluster list is cl_1 whose size is 11 time units which is larger than the threshold. Hence it cannot be merged despite having dependencies with other clusters. Next in the list is cl_2 which has one dependency with its highest connected cluster cl_3. Their combined size is 4 which is below the threshold and they are merged into cl_{23}. Next cl_4 remains in a separate cluster as it has no dependencies with cl_{23} and it cannot be merged with cl_1 due to the threshold.

Algorithm 4: LoadBalancedAllocation()

Input : CL_{merged}, R
Output: CA

1 $\forall r \in R, capacity(r) := \sum_{cl \in CL_{merged}} sz(cl)$;
2 CL_{Sorted} := Clusters sorted on decreasing size;
3 **for** $cl \in CL_{Sorted}$ **do**
4 $r := \arg\max_{r \in R} capacity(r)$
5 $CA := CA \cup (cl, r)$;
6 $capacity(r) := capacity(r) - sz(cl)$;
7 **end**
8 **return** CA;

VI. LOAD BALANCED ALLOCATION

After merging, the clusters are allocated to the available resources while balancing the workload. This step, detailed in Algorithm 4, ensures that clusters are evenly distributed on the cores. The cluster allocation function is defined below.

Definition 6. (CLUSTER ALLOCATION) *A cluster allocation is a function which allocates a cluster in CL to a resource in R. It is denoted by $CA : CL \rightarrow R$.*

At first, all resources in R are assumed to have maximum capacity equal to the sum of the sizes of all clusters in line 1. Clusters are sorted in decreasing order of size in line 2. The largest cluster is then allocated to the resource having the highest remaining capacity in lines 3-5. The size of the cluster is then deducted from the capacity of the chosen resource in line 6 and the process repeats until all clusters are allocated. When a cluster is allocated to a resource $r \in R$, all tasks in the cluster are bound to r. This algorithm has linear complexity. The overall complexity of CMA comes from the merging step and is equal to $O(|T|(|T| + |D|))$. In the example, the allocation step assigns cl_1 to one resource and cl_{23} together with cl_4 to the other resource. This gives us the final binding where tasks a_1, a_2, a_4, a_5 are bound to one processor and tasks a_3, a_5, a_7, a_8 to the other.

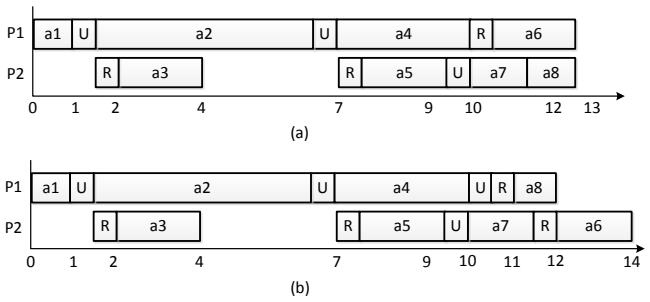

Fig. 4. Schedules obtained from (a) CMA, (b) BDSC for the example DAG

VII. Experimental Setup and Results

In this section, we evaluate CMA by comparing its binding to BDSC on industrial test cases and other well-known test cases. For the evaluation, we need the scheduler to order the tasks once they are bound to resources. We first elaborate on this scheduler setup.

A. Scheduler Setup

We adopt the list scheduler with the earliest due-date first (EDDF) heuristic from [7]. It has been statistically shown in [7] that the EDDF heuristic outperforms classical methods in obtaining feasible schedules for DAGs with deadlines and fixed task bindings. A list scheduler maintains a list of enabled tasks at each step of scheduling. A task is enabled when its predecessors have completed execution. It selects the task with the earliest due-date from the enabled tasks to be scheduled next. Due-dates are computed using an efficient and linear technique that exploits the binding of tasks to get task completion time bounds. By computing these due-dates after the binding technique presented in this paper, we can exploit the task bindings in the due-date computations and get tight bounds on the completion times.

The list scheduler presented in [7] does not take communication overhead into account to compute start times of tasks. A small modification in the computation of the start times is needed to accommodate the communication overhead. We perform transitive reduction at each scheduling step based on the partial schedule to estimate the number of synchronization operations to be taken into account in the start times using Equation 4.

$$s_t = \max(\max_{p \in pred(t)} c_p, \max_{t' \in T \& (r_{t'} = r_t)} c_{t'}) + tranRedCommCosts \tag{4}$$

Here, $\max_{p \in pred(t)} c_p$ and $\max_{t' \in T \& (r_{t'} = r_t)} c_{t'}$ give the completion time of the last completing predecessor of t and the last task scheduled on r_t respectively and $tranRedCommCosts$ refers to the transitively reduced synchronization costs.

For the DAG in Figure 3, the resultant schedule after the ordering of tasks by the scheduler and the transitive reduction of the redundant synchronization is given in Figure 4(a). It has a makespan of 12.5 time units. Figure 4(b) gives the schedule obtained from BDSC which has a makespan of 14 time units.

B. Results

We now compare CMA to BDSC on some industrial and other well known application graphs. We use the scheduler of the previous section for both methods and evaluate the binding based on the resultant makespans and deadlines being met. The features that we use to categorize the test cases are as follows:

- Communication to computation ratio (CCR): This is the ratio of the communication cost to the average computation cost (total of task execution times divided by the number of tasks). A low CCR value implies that the application is computation-intensive. We have generated test cases for CCR values in $\{0.1, 0.5, 1, 2, 3, ..., 10\}$ by adapting the task execution times.
- Number of processors (#P) the application is to be bound to. We have chosen a range of 2-16 processors.
- Size of the application: We quantify application size in terms of its number of tasks.

Industrial applications. We applied the binding approach on three large control applications of the ASML wafer scanners. The architecture of the platforms consists of homogenous general purpose single-core and octo-core processors. Tasks on the octo-core processors use the shared memory communication mechanism between cores. Communication between processors is carried out through a communication network. Due to the relatively high cost of communication through the network compared to shared memory, partial binding of the application tasks on the processors is manually decided by the designers to keep the communication through the network minimal. The remaining binding of the tasks on the cores of the multi-core processors is to be computed using CMA. One of the cores of the octo-core processors is reserved for background processing implying that tasks bound to the octo-cores can choose from the remaining seven cores. Additionally, the tasks in these applications are subdivided into operational phases based on criticality. If there exists a dependency from phase A to B, then all tasks in phase A should complete execution before the beginning of phase B. Hence, we have performed the binding of the phases separately and reported the final makespan results. Some phases do not have much parallelism to be exploited; others have very few and relatively independent tasks. These corner cases have been excluded from binding. During scheduling, the scheduler binds tasks in the excluded phases to the core with the earliest start time. Synchronization cost is fixed to $1.95 \cdot 10^{-9}s$ for the ASML platform to abstract from jitter and caching delay variations. Task execution times and synchronization costs are taken as the average of measurements. The applications are:

1) *Wafer Stages.* This application has several components having a large number of sensors and actuator tasks involved in the accurate movement and positioning of silicon wafers in the scanners. The control tasks range from simple additions to more complex filtering, clipping and matrix operations. We took the wafer stage application from the NXE version of the TWINSCAN wafer scanners of ASML. It has 4,438 control tasks and 48,491 dependencies, with a degree (total incoming and outgoing dependencies) ranging from 1 to 240. The application graph is repeatedly executed at a sample frequency of 20kHz. This frequency translates to a latency requirement of $5 \cdot 10^{-5}s$ which is taken to be the available budget on the processor. The platform consists of eight octo-cores. Task deadlines, decided based on their criticality by domain experts in ASML, range from 50% to 100% of the processor budget. Task execution times range from $5 \cdot 10^{-10}s$ to $2.2 \cdot 10^{-5}s$ with an average of $1.1 \cdot 10^{-7}s$. The CCR values range from 0.009 to 383 with an average of 11. CMA produces a makespan of $3.61 \cdot 10^{-5}s$. This is 3% lower than the $3.73 \cdot 10^{-5}s$ obtained from BDSC. The completion times of the last completing tasks per multi-core processor from CMA are 0-9% lower than BDSC.

2) *FlexRay.* FlexRay is a component that consists of thou-

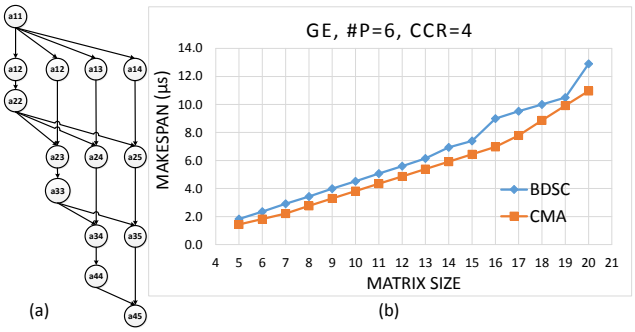

Fig. 5. (a) Example graph for GE, (b) GE results for the given matrix sizes

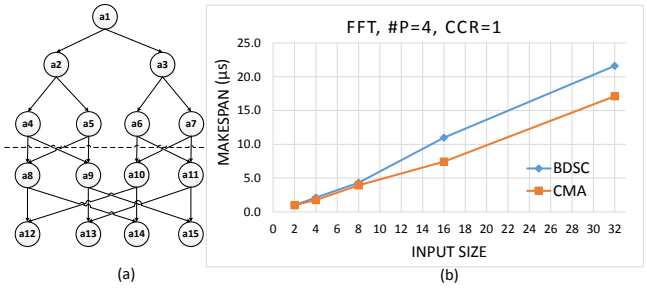

Fig. 6. (a) Example graph for FFT, (b) FFT results for the given input sizes

sands of mirrors that work in parallel to transmit light beams precisely onto the wafers during the lithography process. It is one of the largest applications in the wafer scanner consisting of 13,950 tasks and 663,141 dependencies, with the degree of the tasks ranging from 1 to 1,233, executed at a sample frequency of 238Hz. The platform consists of four octo-core processors. Task deadlines range from 50% to 100% of the processor budget. Task execution times range from $1.8 \cdot 10^{-8}s$ to $1.5 \cdot 10^{-6}s$ with an average of $1.9 \cdot 10^{-7}s$. The CCR values range from 0.1 to 11 with 1.4 being the average. CMA produces a makespan of $3.41 \cdot 10^{-3}s$ which is 0.6% lower than the $3.43 \cdot 10^{-3}s$ from BDSC. The completion times of the last tasks per multi-core processor are lower by 0.3% to 14.7%.

3) *Projection optics box (POB).* The projection optics box (POB) is the enclosure around the optical mirrors that are used to expose the wafers in lithography. It is a smaller application consisting of 2,769 tasks and 35,617 dependencies, with a degree in the range 1-158. The application is composed of two independent parts. One part is executed at a sample frequency of 20kHz and bound to three octo-core processors. The other is executed at 10kHz and bound to two single-core processors. Deadlines of tasks range from 50% to 100% of their respective processor budgets. Task execution times range from $2.5 \cdot 10^{-10}s$ to $1.8 \cdot 10^{-6}s$ with an average of $1 \cdot 10^{-7}s$. CCR values range from 0.1 to 774 with an average of 20. The makespan of $2.16 \cdot 10^{-5}s$ from CMA is 6% lower compared to the $2.31 \cdot 10^{-5}s$ from BDSC. The completion times of the last tasks on the multi-core processors are lower by 5-9%.

In all of the above applications task deadlines were met by both approaches. However, CMA produces a lower makespan in each case. As we know that BDSC only focuses on makespans and does not account for task deadlines, we need another experiment to make a fairer comparison focusing on makespan results alone. We henceforth abstract from deadlines by assigning equal and very large deadlines to all tasks.

Application graphs of well-known problems. We generated

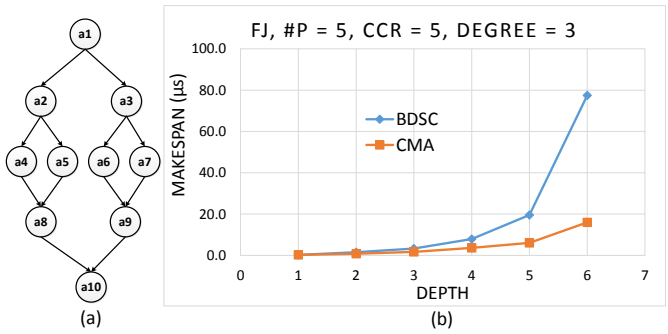

Fig. 7. (a) Example graph for FJ, (b) FJ results for the given depth values

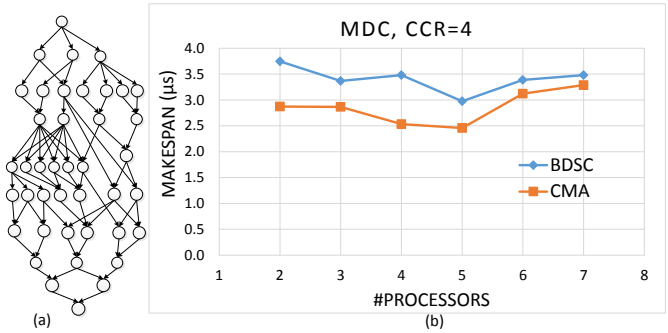

Fig. 8. (a) MDC task graph, (b) MDC results over the number of processors

applications graphs of varying sizes bound to varying sized platforms for four kinds of well-known problems. For each case, we chose to show a result plot for one set of input values that is representative of the typical outcome observed.

1) *Gaussian Elimination (GE).* Gaussian elimination is an algorithm that solves a matrix of linear equations to return the values of the unknown variables [8]. For a matrix of size m, the corresponding task graph has $\frac{m^2+m-2}{2}$ tasks. Figure 5(a) is the task graph for a matrix size of 5. The critical path is $a_{11}, a_{12}, a_{22}, a_{23}, a_{33}, a_{34}, a_{44}, a_{45}$ having the most tasks. We generated a wide range of test cases for matrix sizes from 5 to 20. The outcome for a platform of 6 processors and a CCR of 4 over all matrix sizes is in Figure 5(b). We see that CMA always produces a lower makespan than BDSC.

2) *Fast Fourier Transform (FFT).* Fast fourier transform [9] is an algorithm that computes the Discrete Fourier Transform of a sequence of equally spaced samples of a signal. The task graph corresponding to an input consisting of 4 data points is in Figure 6(a) [10]. Any path starting from the source task to any of the leaf tasks is a critical path. For an input of size m where $m = 2^k$ for some integer k, there are $2m - 1 + m \cdot log_2 m$ tasks in the graph. We have generated test cases for input sizes 2, 4, 8, 16, 32. Figure 6(b) gives the outcome for 4 processors and CCR of 1 for all input sizes. CMA and BDSC produce similar makespans for smaller graphs, with CMA giving smaller makespans for larger graphs.

3) *Fork Join Graphs (FJ).* These graphs consist of a series of fork operations starting from a source node that are then followed by join operations to a leaf node. We use the depth and degree parameter to define these graphs. The depth specifies the number of recursive fork operations and degree specifies the number of children of a task produced during a fork. Figure 7(a) is a fork-join graph for a depth and degree of 2 [11]. Again, any path starting from the source task to

TABLE IV
MAKESPAN COMPARISON RESULTS OF CMA AND BDSC FOR THE WELL-KNOWN APPLICATIONS

Application	Lower m_{BDSC}	Lower m_{CMA}	Equal
GE	638	2185	57
FFT	77	537	286
FJ	373	2203	484
MDC	19	53	0

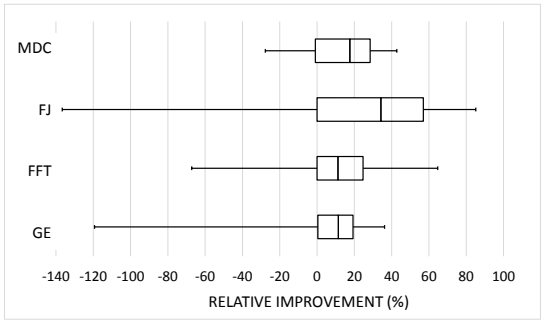

Fig. 9. Box plots of relative percentage improvements of CMA over BDSC

the leaf task is critical. We generated test cases with degree in the range 2-4 and depth in 1-6. Figure 7(b) is the result for 5 processors, a CCR value of 5 and a degree of 3 for the different depth values. The difference in makespans for higher depths is more magnified than in the other test cases. This is because the size of the fork join graphs with depths of 4-6 are much larger than the largest graphs of the other test cases. Similar results are also seen for changing degrees. This shows that our algorithm gives higher gains for larger graphs with more communicating tasks.

4) *Molecular Dynamics Code (MDC).* Molecular dynamics is a method of computer simulation that is used to study physical movements of atoms and molecules. The graph of its code, in Figure 8(a) [10], has a fixed number of tasks and a relatively irregular structure compared to the other test cases which makes it also useful for evaluating our algorithm. We vary the CCR values and the number of processors (from 2 to 7) per test case. The outcome for CCR of 4 is given in Figure 8. CMA produces lower makespans than BDSC and we observe that the makespan difference reduces as we add resources, showing the benefit of using CMA for limited resources. Another interesting observation is that more processors does not necessarily imply smaller makespan. It is possible that with more processors the amount of synchronization needed among cores increases thus adversely effecting the makespan.

Table IV summarizes the overall results of the above test cases. The first column lists the application type. The second and third columns give the number of test cases where BDSC and CMA respectively produce the lowest makespan. The last column gives the number of equal makespan cases. We see that CMA produces lower makespans than BDSC in a significantly higher number of test cases of each type. The box-plot of the percentage of improvements per case are given in Figure 9. The leftmost and rightmost end of the horizontal line give the lowest and highest improvement respectively. The left and right ends of the box are the lower and upper quartiles and the line dividing the box is the median. The typical improvements lie in the range between the lower and upper quartile covering 50% of the cases. This range is 0.5-19% for GE, 0-25% for FFT, 0-57% for FJ and -0.9-28% for MDC which is mainly

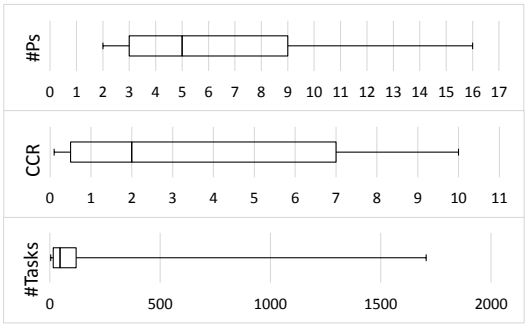

Fig. 10. Box plot of (a) number of processors, (b) CCR values, and (c) number of tasks in the graph for test cases where CMA produces higher makespans

positive. To gain insight in the cases where BDSC results in better makespan, we have the box-plots of those cases in terms of number of processors, CCR and graph sizes in Figure 10. The graphs where BDSC performs better are typically smaller. This highlights the usefulness of CMA for larger graphs. Another observation is that the negative extremes (<-50%) for FJ and GE in Figure 9 are either small graphs with CCR of above 8 or bigger graphs bound to 2 processors. Both cases are outliers in Figure 10 implying that the negative extremes are due to a few corner cases that are not typical.

We also performed experiments by adapting CMA to message passing systems (using the $tlevel$ computation of DSC) and observed that CMA and BDSC are similar in terms of the number of test cases where they perform best. This implies that CMA is also applicable to message passing systems. However, it confirms our hypothesis that the three-step approach of CMA is particularly beneficial when dealing with the inaccuracy in timing computations in the shared memory case.

VIII. RELATED WORK

Standard combinatorial optimization techniques such as branch and bound algorithms [12], tabu search [13] and simulated annealing [14] search the state-space for optimal solutions with different heuristic search strategies. To achieve good quality the algorithms rely on the exploration of a significant number of alternative solutions. This works very well for small to medium sized graphs in a reasonable amount of time. Our solution targets solutions for large DAGs constructing only a single solution. We therefore resort to list scheduling [6] and clustering [2] based methods that are efficient and commonly used in DAG binding and scheduling.

The clustering step in our work adopts the DSC algorithm in [2] and modifies it to deal with individual task deadlines and the shared memory communication mechanism. DSC has been shown to outperform other clustering methods for an unbounded number of processors in terms of results and algorithmic complexity [3]. DSC performs clustering assuming unlimited resources. We compare CMA to BDSC, an extension of DSC for limited resources, that uses a common binding method for both shared memory and message passing systems. It uses the justification that if it is worthwhile to nullify a communication cost for message passing systems, it is also worthwhile for shared memory systems. CMA, on the other hand, explicitly takes the shared memory communication framework into account in its extension of DSC that enables it to make better clustering decisions. Its three-step approach uses more high level structural information in the graph compared to the local decisions made by BDSC. The algorithmic

complexity of BDSC is $O(|T|^3)$, which is the same as the complexity of CMA.

One of the earliest methods to consider platform size limitations, proposed by Sarkar in [15], performs clustering assuming unlimited resources and then uses an incremental approach to perform the assignment of clusters to resources iteratively. Each iteration selects and assigns a task along with all other tasks in the cluster to the processor that gives the least makespan increase from the last iteration. This needs makespan computation by scheduling all the tasks in the cluster for each processor per iteration. Its complexity of $O(|T||R|(|T|+|D|))$ makes the algorithm slow for large graphs. A low complexity binding approach in PYRROS [16] first uses DSC to obtain the initial clusters. It then sorts the clusters in decreasing order of their loads and then maps them to the processors in a balanced manner. Such a mapping risks assigning independent clusters onto the same resource while separating dependent clusters onto different resources which can increase the makespan. CMA reduces this risk by performing an intermediate merging step based on dependencies. Also CMA, in its first step, adapts the clustering of DSC to deal with shared memory systems.

Another approach for limited processors, called Triplet [17], also adopts a three-step approach for binding to a system of heterogenous systems called workstations. The first clustering step on unlimited resources only uses task top levels to sort tasks. CMA utilizes more information by taking top levels and due-dates to sort tasks during clustering. After clustering, Triplet clusters the workstations to form homogenous workstation clusters. Lastly, clusters are sorted on the amount of external communication and workload and assigned to the workstation cluster that gives the earliest start time. In CMA's second step, we sort clusters on the number of dependencies with their highest connected cluster. We then merge the highest connected clusters together thus removing the communication cost between them. Only then we allocate the clusters to processors. This second step allows us to further reduce communication between clusters thus minimizing makespans.

An approach for shared memory systems and limited resources that considers the concept of synchronization time, like in CMA, is the Extended Latency Time (ELT) [18] approach. It calculates priorities of tasks using a combination of metrics including top levels and bottom levels. It then performs binding by choosing tasks based on the priority order, while keeping the priorities fixed. CMA, like DSC and BDSC, uses dynamic priorities which are updated during clustering to account for the edge costs that become zero when two tasks are clustered together. This helps in making better binding decisions as it can detect changes in the critical paths that can arise due to zeroing of edge costs, also pointed out in [19].

Other single-step binding algorithms include High Level First with Estimated Times [20], Heterogenous Earliest-Finish Time [10], and Constrained Earliest Finish Time [11]. In [3], BDSC is shown to outperform these as well as the earlier mentioned multi-step approaches. BDSC is thus chosen as our benchmark for comparison. This paper shows that CMA outperforms BDSC for binding in shared memory systems.

Binding algorithms typically focus on makespan reductions and do not take task deadlines into account. An approach that does consider deadlines is given in [21]. However, it considers independent sporadic tasks and is preemptive also allowing task migration, none of which are applicable to our problem domain. CMA performs binding of DAGs on multiprocessor platforms by utilizing parallelism while taking both communication overhead and task deadlines into account.

IX. Conclusions

We presented a three-step binding algorithm that computes the binding of tasks with deadlines on the limited resources of a shared memory multiprocessor platform. The first step, a deadline aware shared memory extension of the DSC algorithm, produces a clustered graph assuming unlimited resources. Since there can be more clusters than resources, the next step performs merging of clusters based on dependencies while constraining them to be smaller than a threshold. Finally the clusters are allocated to the resources by balancing the workload. CMA has been validated by evaluating its impact on our earliest due-date first scheduler on large control applications of ASML. CMA produces lower makespans in comparison to the state of the art BDSC algorithm. It also outperforms BDSC in a high number of test cases of other well-known parallel problems.

References

[1] "ASML," http://www.asml.com, accessed: 18/01/2016.

[2] T. Yang and A. Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, Sep 1994.

[3] D. Khaldi, P. Jouvelot, and C. Ancourt, "Parallelizing with BDSC, a resource-constrained scheduling algorithm for shared and distributed memory systems," *Parallel Computing*, vol. 41, pp. 66 – 89, 2015.

[4] A. V. Aho, M. R. Garey, and J. D. Ullman, "The transitive reduction of a directed graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972. [Online]. Available: http://dx.doi.org/10.1137/0201008

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[6] T. C. Hu, "Parallel sequencing and assembly line problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961.

[7] S. Adyanthaya, M. Geilen, T. Basten, R. Schiffelers, B. Theelen, and J. Voeten, "Fast multiprocessor scheduling with fixed task binding of large scale industrial cyber physical systems," in *Euromicro Conference on Digital System Design (DSD)*, Sept 2013, pp. 979–988.

[8] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, "Parallel gaussian elimination on an mimd computer," *Parallel Computing*, vol. 6, no. 3, pp. 275 – 296, 1988.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[10] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[11] M. A. Khan, "Scheduling for heterogeneous systems using constrained critical paths," *Parallel Computing*, vol. 38, no. 45, pp. 175 – 193, 2012.

[12] J. Carlier and E. Pinson, "An algorithm for solving the job-shop problem," *Management Science*, vol. 35, no. 2, pp. 164–176, 1989.

[13] F. Glover, "Tabu search – part i," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190–206, 1989.

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[15] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[16] T. Yang and A. Gerasoulis, "PYRROS: Static task scheduling and code generation for message passing multiprocessors," in *Proceedings of the 6th International Conference on Supercomputing*, ser. ICS '92, 1992, pp. 428–437.

[17] B. Cirou and E. Jeannot, "Triplet: A clustering scheduling algorithm for heterogeneous systems," in *International Conference on Parallel Processing Workshops*, 2001, pp. 231–236.

[18] M. Solar and M. Inostroza, "A scheduling algorithm to optimize real-world applications," in *Distributed Computing Systems Workshops. Proceedings. 24th International Conference on*, March 2004, pp. 858–862.

[19] Y. K. Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, Mar 1998, pp. 531–537.

[20] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Communications of the ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974.

[21] S. Baruah and B. Brandenburg, "Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, Dec 2013, pp. 160–169.