

# Real-World Clustering for Task Graphs on Shared Memory Systems

Alexander Herz<sup>(✉)</sup> and Chris Pinkau

Lehrstuhl Für Informatik II/XIV, Technische Universität München,  
Boltzmannstraße 3, 85748 Garching b. München, Germany  
`{herz,pinkau}@in.tum.de`

**Abstract.** Due to the increasing desire for safe and (semi-)automated parallelization of software, the scheduling of automatically generated task graphs becomes increasingly important. Previous *static* scheduling algorithms assume negligible run-time overhead of spawning and joining tasks. We show that this overhead is significant for small- to medium-sized tasks which can often be found in automatically generated task graphs and in existing parallel applications.

By comparing real-world execution times of a schedule to the predicted static schedule lengths we show that the static schedule lengths are uncorrelated to the measured execution times and underestimate the execution times of task graphs by factors up to a thousand if the task graph contains small tasks. The static schedules are realistic only in the limiting case when all tasks are vastly larger than the scheduling overhead. Thus, for non-large tasks the real-world speedup achieved with these algorithms may be arbitrarily bad, maybe using many cores to realize a speedup even smaller than one, irrespective of any theoretical guarantees given for these algorithms. This is especially harmful on battery driven devices that would shut down unused cores.

We derive a model to predict parallel task execution times on symmetric schedulers, i.e. where the run-time scheduling overhead is homogeneous. The soundness of the model is verified by comparing static and real-world overhead of different run-time schedulers. Finally, we present the first clustering algorithm which guarantees a real-world speedup by clustering all parallel tasks in the task graph that cannot be efficiently executed in parallel. Our algorithm considers both, the specific target hardware and scheduler implementation and is cubic in the size of the task graph.

Our results are confirmed by applying our algorithm to a large set of randomly generated benchmark task graphs.

**Keywords:** Static scheduling · Run-time overhead · Execution time prediction

## Introduction

The diminishing clock speed gains realized in new processor designs and fabrication processes have produced a rise of multi- and many-core CPUs for servers,

desktops and even embedded devices. This development increases the pressure to produce parallel software and schedules for this software which are efficient in terms of overall execution time and power consumption, especially for battery driven devices. Due to the NP-completeness of many instances of the scheduling problem, a set of heuristics trying to approximate the best solution have been proposed. Kwok [14] and McCreary [13] have compared the quality of a range of well-known heuristics in terms of the static schedule length predicted by the different scheduling heuristics for a versatile set of input task graphs. For some heuristics (e.g. linear clustering [8]) it has been proven that the produced schedule length (the makespan as predicted by the heuristic) is no longer than the fully sequential schedule. For large-grain task graphs it has been shown that the schedule lengths from greedy algorithms are within a factor of two of the optimal schedule [11].

To the best of our knowledge, no comparison of static schedule lengths to the real-world execution time of the schedule has been undertaken. If the real execution time of a schedule on a specific target platform does not at least roughly correlate to the static schedule length, then any guarantees or schedule length advantages of one heuristic compared to another are purely theoretical as the static schedule does not model reality.

Developments in the research community show that automatic parallelization is an important part of the future of computer science. Implicitly parallel compilers automatically extract task graphs from user programs. Typically, the extracted task sizes are small [9] because all computations are considered for parallel execution. Therefore, the proper scheduling of small-grain task graphs is fundamental for automatic parallelization.

In contrast to a Gantt-chart’s implication that tasks are started at a predefined time, most parallel systems (e.g. [12,21]) implement a dynamic signaling mechanism to spawn and join tasks as soon as all preconditions are satisfied. This removes the burden of providing hard real-time guarantees for the software and hardware which may produce unnecessary long schedules as worst case estimates must be used everywhere.

Our measurements show that on some platforms the overhead is in the order of  $2 \cdot 10^4$  [clocks] so that it can be ignored only in the limiting case when all tasks are in the order of  $10^6$  [clocks] and larger. The maximum possible task size for a game or numerical simulation running at 60 frames per second on 2 GHz CPUs is in the order of  $10^7$  [clocks] and typically much smaller for non-trivially parallelizable problems. This shows that the overhead is relevant for typical parallel applications. For task graphs that contain tasks with sizes in the order of the overhead, traditional scheduling algorithms may produce schedules with arbitrarily bad speedup (e.g. a speedup smaller than one on more than one core compared to the fully sequential schedule) as they ignore the run-time overhead (cost of spawning and joining tasks). This includes algorithms that in theory guarantee a schedule length shorter than the fully sequential schedule. The insufficient speedup produced by these algorithms is especially harmful on battery driven devices that could shut down cores that are used inefficiently.

Furthermore, the neglected overhead may shift data ready times used to perform scheduling in most algorithms asymmetrically, so that tasks which appear to run in parallel for the algorithm will not run in parallel in reality.

Our main contributions to solve these problems are:

- We derive a generic model to predict run-time scheduler overhead for symmetric schedulers, i.e. the scheduling overhead is homogeneous.
- We show that our model accurately predicts scheduling overhead for stealing [4] and non-stealing schedulers on different hardware.
- We define a task granularity for communication-free task graphs related to the parallel task execution times on a specific platform.
- We present the first clustering algorithm that guarantees a minimum real-world speedup per core for communication-free task graphs.

The rest of the paper is structured as follows. In Sect. 1 we show that the measured execution times of a simple task graph have a non-linear relationship to the static schedule length from traditional scheduling algorithms that ignore the scheduling overhead. Afterwards, in Sect. 2, we derive a statistical model to predict scheduling overhead for symmetric schedulers where the run-time scheduling overhead is homogeneous. This is followed by Sect. 3, where we show that our overhead model accurately predicts the scheduling overhead on several platforms and scheduler implementations. The model is used in Sect. 4, to construct a clustering algorithm which guarantees a user defined minimum speedup per core in  $O(n^3)$ . Section 5 presents benchmarks showing how our algorithm improves the average speedup of a large set of randomly generated task graphs with small and large task sizes. Finally, we discuss related work and our results in Sect. 6, as well as possible future work in Sect. 7.

## 1 Example

We will examine the task graph in Fig. 1.

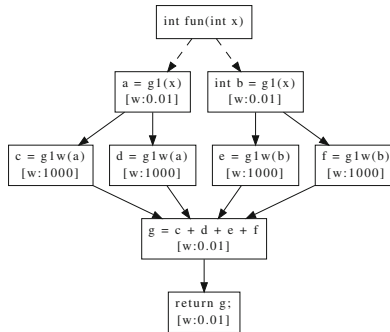
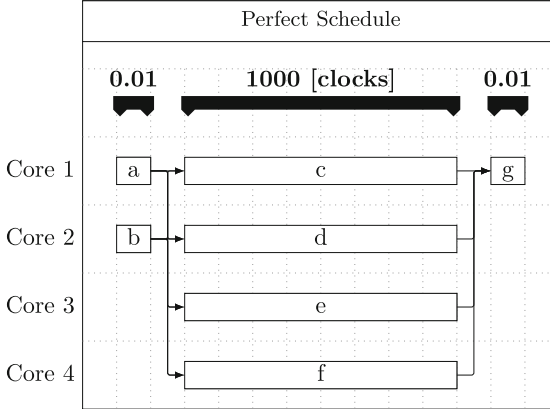


Fig. 1. Example task graph containing small tasks.

Scheduling the program depicted in Fig. 1 on multi-core hardware appears trivial. Given a machine with more than four cores, perfect scheduling [6] can be applied to produce the best possible schedule, according to the scheduler’s model of program execution, in  $O(n + m)$ . This may produce the Gantt-chart shown in Fig. 2.



**Fig. 2.** Gantt-chart for task graph in Fig. 1 produced using perfect scheduling [6]. Arrows show dependencies but do not require any time according to the scheduling algorithm. The predicted sequential execution time of 4000 clocks compared to the predicted parallel execution time of 1000 clocks suggests a speedup of about 4. Actually executing the task graph yields a measured execution time of ca. 4000 clocks on a 4 Core Nehalem for the fully sequential program and a measured execution time of 6000 clocks for the fully parallel program. In contrast to the prediction shown in the Gantt-chart, sequential execution is about 1.5 times faster than parallel execution, not 4 times slower.

Although the Gantt-chart implies that tasks may start execution at a specified time this is rarely implemented. In order to start tasks based on the times from the chart, hard real-time constraints need to be placed on the executing hard- and software and sound worst case execution times for all tasks must be calculated. In addition, the executed schedule might be less than optimal as some tasks may be able to execute before the worst case finish time of their predecessors because the predecessors finished earlier than the conservative estimate.

Most scheduler implementations (e.g. [12, 21]) signal waiting tasks as soon as all their preconditions have been fulfilled so that they can start executing as soon as possible. Usually, spawning tasks without preconditions also requires to signal to another thread.

The Gantt-chart in Fig. 2 suggests a parallel execution time of about  $0.01 + 1000 + 0.01 = 1000.01$  clocks, so a speedup of about 4 compared to the sequential execution of ca.  $2 \cdot 0.01 + 4 \cdot 1000 + 0.01 = 4000.03$  clocks is predicted. Actually running the program in its fully parallel version using TBB’s [12] stealing scheduler takes about 6000 clocks per task graph execution on a 4 Core Nehalem.

In contrast, executing the sequential version of the program on the same hardware yields an execution time of ca. 4000 clocks which is about 1.5 times faster than the parallel version.

Apparently, the existing scheduling algorithms do not model the real-world scheduling process, but an artificial scheduler that has no run-time overhead and always yields linear speedup. This leads to unrealistically small execution time predictions from these scheduling algorithms. The existing literature does not define a grain size for task graphs without communication costs (and an unrealistic one for small tasks with small communication costs). In Sect. 2 we define such a grain size in direct connection to the parallel execution time of a task on a specific platform.

In Sects. 3 and 5, it will be shown that the scheduling overhead is not negligible on real-world systems even for bigger task sizes up to  $10^5$  clocks and more (depending on the specific hardware and scheduler implementation). This emphasizes that the effect is relevant for normal task graphs that do not contain minuscule tasks.

## 2 Model

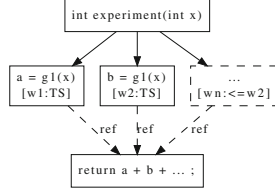
In this section, a model for the prediction of task execution times that accounts for the scheduling overhead is derived, improving the execution time prediction accuracy by up to 1000 % (compared to previous scheduling algorithms which neglect the scheduling overhead for smaller tasks).

Since the actual execution times on real hardware fluctuate heavily depending on the overall system state, a stochastic model is developed. Adve and Vernon [1] have found that random fluctuations have little impact on the parallel execution time, so that the expectation values we derive should be a good model of the real execution time. First, we derive the model for a stealing scheduler as implemented in Thread Building Blocks 4.1 (TBB [12]). Then we generalize the model to schedulers with symmetric scheduling overhead, i.e. the overhead is homogeneous.

### 2.1 Two Node Fork/Join Graphs

In the following, the execution of fork/join task graphs with two tasks are modeled. After deriving a model for two tasks, we will extend the model for more tasks.

The work weight associated with each task represents the average execution time of the task on the target platform including all costs (i.e. resource contention, cache effects, etc.) except for the dynamic scheduling costs. Without loss of generality, we assume that the task calculating  $a$  (from Fig. 3 with  $n = 2$ ) is executed locally after the second task calculating  $b$  has been spawned at time  $t_0$ . For stealing schedulers, spawning means that the task is enqueued in a list on the thread where the task is spawned. If no other idle core steals the task from this list for parallel execution, the task will eventually be executed



**Fig. 3.** Generic fork/join task graph with  $n$  nodes. Here  $g1$  performs a trivial loop that takes  $TS$  clocks and does not interfere with the other tasks. The tasks are spawned by the root node and the last task is notified as soon as all predecessor tasks have finished.

locally on the thread that originally spawned the task. The spawned task can be stolen (and hence be executed in parallel) from the time it was spawned up until the first task finished execution at  $t_0 + TS$  (where  $TS = \max\{w_1, w_2, \dots, w_n\}$  is the maximum size of all tasks because the largest task determines the overall execution time) and starts to execute the spawned task itself. For convenience, we set  $t_0 = 0$ .

First, the time to spawn a task on another core is calculated.

Since the stealing process is independent from the spawn (other idle cores check regularly if there is something to steal), the normalized probability that a steal attempt is made at time  $t$  is

$$p_{\text{attempt}} = \frac{\sigma}{TS}, \quad (1)$$

where  $0 < \sigma < 1$  determines the steal attempt frequency which depends on the actual scheduler and hardware. The frequency is smaller than one because less than one attempt per clock can be made on real systems. A steal attempt need not succeed, e.g. if there is nothing to steal. The probability that the available task was not stolen until time  $t$  is given by the inverse of the probability that it was stolen:

$$p_{\text{not-stolen}}(t) = 1 - \int_0^t p_{\text{attempt}} dt = 1 - \frac{\sigma}{TS} \cdot t. \quad (2)$$

So the probability that a steal attempt is successful at time  $t$  is given by the probability that it was not yet stolen times the steal attempt probability (when within the time frame where the task can be stolen at all):

$$p_{\text{stolen}}(t) = \begin{cases} p_{\text{not-stolen}}(t) \cdot p_{\text{attempt}} & 0 \leq t \leq TS \\ 0 & \text{else.} \end{cases} \quad (3)$$

Finally, the expected time  $T_{\text{steal}}$  for the second task being stolen is

$$T_{\text{steal}} = \int_0^{TS} p_{\text{stolen}}(t) \cdot t dt = \frac{(3\sigma - 2\sigma^2)}{6} \cdot TS =: \beta \cdot TS. \quad (4)$$

The overall parallel execution time (PET) of the potentially stolen task is given by

$$pet(TS) = \beta \cdot TS + \alpha' + TS. \quad (5)$$

Here,  $\alpha'$  is the expectation value of the fixed overhead required to execute the steal (which is initiated at time  $T_{\text{steal}}$ ) and the join to wait for both tasks. After all overhead is accounted for, the time  $TS$  needed to execute the task must be added. Like  $\sigma$ ,  $\alpha'$  depends on the hard- and software and must be obtained by running an experiment on the specific target platform.

The speedup achieved when running all tasks in parallel is obtained via

$$su(w_1, \dots, w_n) = \frac{\sum_i^n w_i}{pet(\max\{w_i\})}, \quad (6)$$

where  $\sum_i^n w_i$  is the sequential execution time (SET).

For non-stealing schedulers (e.g. MPI [21] based code), the derivation is essentially identical. The tasks are signaled rather than stolen. The signaling is implemented via `MPI_send` and `MPI_receive` or `MPI_barrier` so that only negligible data sizes are communicated, transmission of larger amounts of data is not modeled as the article's scope is limited to cost-free communication. Hence,  $p_{\text{attempt}}$  becomes the probability that the signal starting the second task arrives at time  $t$ . In addition, the second task can be signaled long after the first finished executing when a non-stealing scheduler is used. Assuming there exists a maximum time  $T_{\text{max}} > TS$ , after which the second task is guaranteed to have been signaled, we substitute  $T_{\text{max}}$  for all  $TS$  in Eqs. 1 to 4 to obtain the expectation time that the signal arrives  $T_{\text{signal}} = \beta \cdot T_{\text{max}}$ . Rewriting this with  $T_{\text{max}} = TS + \delta T$  we get

$$T_{\text{signal}} = \beta \cdot TS + \beta \cdot \delta T. \quad (7)$$

As  $\beta \cdot \delta T$  is a hardware dependent constant we can subsume it into  $\alpha = \alpha' + \beta \cdot \delta T$  and add it to the final parallel execution time prediction by substituting  $\alpha'$  by  $\alpha$  in Eq. 5:

$$pet_{\text{general}}(TS) = \beta \cdot TS + \alpha + TS. \quad (8)$$

The form of the final expression to evaluate the parallel execution time  $pet_{\text{general}}$  of the tasks is independent of the underlying scheduler.

The break even point (BEP) on a specific target platform is defined as the task size  $BEP$  that gives a speedup of 1:

$$su(BEP, BEP) = 1. \quad (9)$$

Finally, granularity for communication-free task graphs is defined as follows. A task is said to be small-grain if its associated work is smaller than the BEP. Conversely, it is considered large-grain if its work exceeds the BEP. A task graph is said to be small-grain if it contains any small-grain tasks.

## 2.2 Many Node Fork/Join Graphs

Fork/Join graphs with more than two spawned nodes as shown in Fig. 3 are handled as follows. The base overhead  $\gamma$  of parallel execution (as determined by the measured execution time of two empty tasks) is subtracted from the

predicted two node execution time  $pet_{\text{general}}$  and divided by the sequential execution time to get the speedup of both tasks compared to the sequential execution. The square root of the combined speedup gives the speedup per task.

$$su_{\text{task}}(TS) = \sqrt{\frac{pet_{\text{general}}(TS) - \gamma}{TS}} \quad (10)$$

The final execution time for a fork/join graph with  $n$  tasks is given in Eq. 11 by applying the speedup per task for every task and adding the base overhead:

$$pet^n(TS) = su_{\text{task}}(TS)^n \cdot TS + \gamma \quad (11)$$

The algorithm presented in Sect. 4 will decompose more complex task graphs into simple fork/join task graphs compatible to Eq. 11 to predict their speedup. In order to apply this model, the target (scheduler and hardware) specific constants  $\alpha$  and  $\beta$  as well as the base overhead  $\gamma$  must be measured.

In the next section, the quality of the model for two task predictions for several different target platforms is evaluated. In Sect. 5, the model for many tasks is applied on a large range of randomly generated task graphs showing that Eq. 11 can accurately predict parallel execution times.

### 3 Verification of the Two Task Model

In order to verify that the model derived in Eq. 5 is sound, the execution time of fork/join task graphs as shown in Fig. 3 executed on several different hardware platforms with different scheduler implementations are measured and compared to the model predictions.

Figure 4 shows that the execution times of tasks of different lengths depend only on the longest task (assuming there are enough hardware resources to execute all tasks in parallel).

Figure 5 shows that the model predictions are in very good agreement with the real behavior of the analyzed hardware and schedulers.

As to be expected, TBB's stealing-based scheduler performs better than the MPI based scheduler for smaller tasks. The BEP is about half the size for the stealing scheduler (2242 vs. 4636 clocks) on the Nehalem<sup>1</sup> platform. The difference is more pronounced on the mobile Sandy<sup>2</sup> platform (6812 vs. 22918 clocks). For the experiments, all MPI processes were placed on the same node, so that no actual network communication was executed.

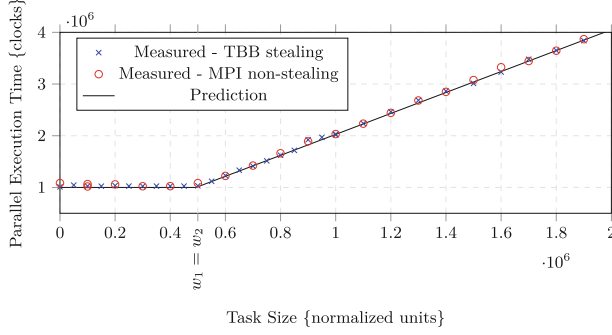
The final values for  $\alpha$  and  $\beta$  for each target platform (hardware and scheduler combination) are obtained by fitting the model from Eq. 6 to the measurements.

TBB's stealing scheduler and MPI's non-stealing task execution differ completely from a conceptual and implementation point of view. As shown in this section, they are both well represented by the model.

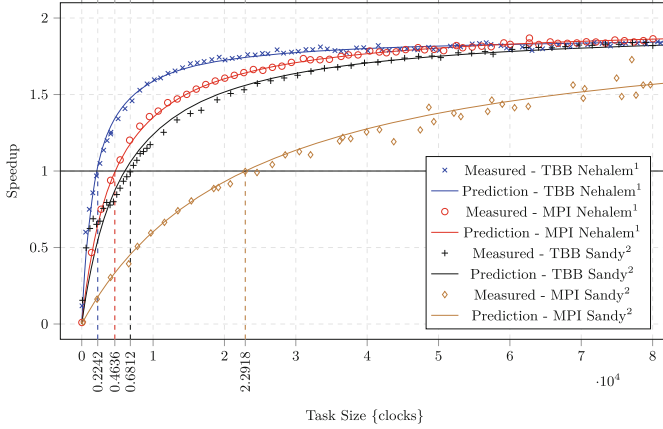
<sup>1</sup> Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz SMP x86\_64 GNU/Linux 3.5.0-37-generic.

<sup>2</sup> Intel(R) Core(TM) i7-3667U CPU @ 2.00GHz SMP x86\_64 GNU/Linux 3.5.0-17-generic.





**Fig. 4.** Comparison of the execution times of two parallel tasks of different size to the prediction that the overall run-time is dominated by the longer task. The first task’s size is increased from 0 to  $2 \cdot 10^6$  while the size of the second task is fixed to  $0.5 \cdot 10^6$  (and vice versa). Error bars have been omitted for readability, all measured data points lie within one standard deviation from the model prediction.



**Fig. 5.** Comparison of the speedup of two parallel tasks with the predicted speedup from Eq. 6. Here,  $\alpha$  and  $\beta$  are determined by fitting the model to the measured data.  $\beta$  is related to the signal probability per time and  $\alpha$  quantifies the delay after the second thread has received the signal until it can start processing the second task. The experiment measures the average time it takes the scheduler to spawn, execute and join two completely independent tasks with equal task size. Error bars have been omitted for readability, all measured data points lie within one standard deviation from the model prediction and are in the order of 10% of the measured value. TBB denotes Thread Building Blocks 4.1 and MPI denotes open MPI 1.4.5. Speedstep was disabled on all systems during measurement to avoid large fluctuations due to the processors power management. The minimum task size required to realize 98% of the possible speedup for the these platforms reaches from  $10^5$  to  $10^6$  [clocks]. For larger tasks, the overhead may be neglected.

In addition, the benchmarks from Sect. 5 will show that Eq. 11 gives a realistic execution time prediction for fork/join graphs with more than two tasks.

In the next section, a preconditioning algorithm is presented that uses the model to merge tasks in a task graph that cannot be efficiently executed in parallel.

## 4 Algorithm

Some previous scheduling algorithms attempt to give guarantees that the produced schedule length is no longer than the fully sequentialized version of the task graph [8] or that the schedule length is within a factor of two from the optimal schedule for task graphs where the computation to communication ratio is high [11]. As will be shown in Sect. 5, scheduling small-grain (w.r.t. task size) task graphs with scheduling algorithms that ignore scheduling overhead (even with perfect scheduling) can lead to surprisingly bad results where the real execution time of the task graph is orders of magnitude worse than the sequential execution.

This shows that the guarantee of the traditional scheduling algorithms is of theoretical nature. The actual execution times and speedups predicted by the traditional algorithms hold only in the limiting case when all tasks sizes are much larger than the BEP. Still, our algorithm improves the speedup even for such large tasks on average by 16% as shown in Sect. 5. Figure 5 shows that close to linear speedup can be expected only for task sizes in the order of tens to hundreds of thousands of clocks or more. This means that the data ready times used to schedule tasks in many traditional algorithms [8] will be underestimated for many tasks and may shift tasks that appear to be parallel by different time offsets so that they will not run in parallel in practice.

In this section we present our clustering and execution time prediction algorithm which preconditions the task graph by collapsing parallel tasks that do not yield a minimum real-world speedup of  $\rho$  per core. The algorithm guarantees that the real-world execution time of the preconditioned task graph (on sufficiently many cores and if it actually contains any parallelism) is strictly smaller than the sequential execution time.

In principle, the algorithm decomposes input task graphs into instances of our modeled fork/join task graph from Fig. 3, applies our model via Eq. 11 and composes the results. This is achieved by the following steps. Due to the Hasse property enforced on the graph, all predecessors of join nodes are parallel nodes. For parallel nodes, the algorithm finds the lowest common ancestor and the highest common descendant, which form a fork/join graph. The model is applied to these fork/join graphs to predict the speedup and merges parallel nodes if the speedup is insufficient. After merging the nodes, the Hasse property is restored. This last step is intuitively done in  $O(n^4)$ , but we present a more elaborate approach to get an upper bound of  $O(n^3)$ .

In the following, the algorithm is presented in several parts. The first part, as seen in Algorithm 1, is the preprocessing part. Here we build several data

structures to make sure that the overall upper bound of  $O(n^3)$  is met. In the second part, see Algorithm 2, the real work is done by calculating the estimated processing times of all tasks and merging appropriate tasks together.

The preprocessing shown in Algorithm 1 computes for all predecessor pairs of join nodes the lowest common ancestor (LCA) with the minimal speedup, as well as the shortest paths from the LCA to the respective pair nodes in the unmerged graph. Later on, the corresponding paths inside the merged graph will be constructed from the paths from the LCA. Transitive edges are removed from the input graph by applying the Hasse reduction, as the algorithm assumes that tasks preceding a join node may execute in parallel which is not true for transitive edges. In addition, all linear task chains are removed from the graph. Both operations reduce the signaling overhead of the graph, as every edge in the graph can be interpreted as a signaling operation for the run-time scheduler.

---

**Algorithm 1.** preprocessing

---

**Require:** graph  $G$

$H \leftarrow \text{HasseReduction}(G)$

$H' \leftarrow H$

calculate APSP

preprocessing for common ancestors

create hashmap  $um$  from unmerged to merged nodes  $um : \{\text{nodes}\} \rightarrow \mathcal{P}(\{\text{nodes}\})$

initialize  $um$ :  $\text{node} \mapsto \{\text{node}\}$

create hashmap  $pp^A$  from pairs of nodes to (lca, path1, path2)  $pp^A : (\text{node}, \text{node}) \rightarrow (LCA, \text{path}, \text{path})$

initialize  $pp^A$ :

**for all** join nodes  $j'$  **do**

**for all** predecessor pairs  $(i', k')$  of  $j'$  **do**

        get the LCA with minimal speedup:  $lca' \leftarrow \text{getLCA}(i', k')$

$p_{i'} \leftarrow \text{shortestPath}(lca', i')$

$p_{k'} \leftarrow \text{shortestPath}(lca', k')$

$pp^A \leftarrow pp^A + \{(i', k') \mapsto (lca', p_{i'}, p_{k'})\}$

**end for**

**end for**

---

Next, the outer loop of the actual algorithm is shown in Algorithm 2. The nodes are visited in a topological order and data ready times are computed from the predecessors unless the current node is a join node which needs the special treatment shown in the **merge** procedure in Algorithm 5.

In the following, the merge operation  $ik \leftarrow i \cup k$  means that a new node  $ik$  is created in the merged graph  $\hat{H}$  that inherits all edges from  $i$  and  $k$  before these nodes are deleted from the graph.

**mergeLinear** is used in order to remove the overhead generated by the communication between consecutive tasks.

**mergeParallel** is used to merge the parallel predecessors of join nodes. If the indegree of the merged node is greater than one then the **merge** procedure is recursively applied to the merged node as it may be a newly created join node.

**Algorithm 2.** outer loop

---

```

while traverse nodes  $j$  in topological order do
  mergeLinear (predecessor of  $j$ ,  $j$ )
  if node  $j$  is a join node then
    merge  $j$ 
    mergeLinear (predecessor of  $j$ ,  $j$ )
  end if
  store estimated starting time  $est$  and estimated finishing time  $drt$  for current node
end while

```

---

**Algorithm 3.** mergeLinear

---

**Require:** task sets  $i$ ,  $k$  to be merged and  $k$  has exactly 1 predecessor

---

```

 $ik \leftarrow i \cup k$ 
for all tasks  $t$  in  $ik$  do
   $um \leftarrow um + \{t \mapsto ik\}$ 
end for
remove Hasse violating edges
update  $nsp$  distances and paths and  $drt$ 

```

---

**Algorithm 4.** mergeParallel

---

**Require:** task sets  $i$  and  $k$  to be merged

---

```

 $ik \leftarrow i \cup k$ 
for all tasks  $t$  in  $ik$  do
   $um \leftarrow um + \{t \mapsto ik\}$ 
end for
remove Hasse violating edges
update  $nsp$  distances and paths and  $drt$  transitively
if indegree  $ik > 1$  then
  merge  $ik$ 
end if

```

---

The general **merge** procedure applied to the join nodes is shown in Algorithm 5.

Here, variables with hat, like  $\hat{H}$ , denote information from the merged graph in its current state, whereas variables with prime, like  $i'$ , denote unmerged information. Variables without hat or prime refer to information from the unmerged graph available from preprocessing. The information from the unmerged graph is translated into the domain of the merged graph using the  $um$  mapping.

In order to obtain realistic task execution times, the available parallel work (fork/join tasks preceding the join node) must be calculated. The algorithm performs this by considering the paths from the lowest common ancestor [3] for each pair of nodes preceding a join node. These paths may be considerably larger than the pair nodes alone. So, for each predecessor of a join node, the longest path from common ancestor to the predecessor node along with the path's start time is stored in the *tasklist*. The *tasklist* is passed to the multift algorithm from Coffman, Garey and Johnson [5] to find a schedule for the parallel tasks preceding the current join node. The multift algorithm uses a  $k$ -step binary

**Algorithm 5.** merge

---

**Require:** join node  $j$   
 create list  $tasklist$   
 $drt_{\min} = \text{Infinity}$   
**for all** predecessors  $i$  of  $j$  in  $\hat{H}$  **do**  
   **for all** predecessors  $k \neq i$  of  $j$  in  $\hat{H}$  **do**  
 unmerged nodes inside merged nodes  
**for all**  $i' \in i, k' \in k$  **and**  $um(i') \neq um(k')$  **and**  $pp^A(i', k')$  exists **do**  
   get the LCA and the corresponding paths from the unmerged graph  $H'$  :  
    $(lca', p_{i'}, p_{k'}) \leftarrow pp^A(i', k')$   
   get the corresponding merged nodes and paths :  $(est_i^p, est_k^p, \hat{p}_i, \hat{p}_k) \leftarrow um(lca', p_{i'}, p_{k'})$   
    $\hat{lca} \leftarrow \text{update}(lca', est_i^p, est_k^p, \hat{p}_i, \hat{p}_k)$ :  $\hat{lca}$  is last common node in  $\hat{p}_i, \hat{p}_k$  and all arguments of update are modified accordingly  
   **if**  $work(\hat{p}_i) > maxwork_i$  **then**  
     get path's run time (includes delays from nodes the path depends on) :  
      $maxwork_i \leftarrow work(\hat{p}_i)$   
      $est_i \leftarrow est_i^p$   
   **end if**  
   calculate  $drt_{\min}$  for common ancestors :  
    $drt_{\min} \leftarrow \min\{drt_{\min}, drt_{\hat{lca}}\}$   
   **end for**  
**end for**  
 $tasklist \leftarrow tasklist + (i, maxwork_i, est_i)$   
**end for**  
 start with maximal parallelism : cores = size of  $tasklist$   
 $C \leftarrow \text{multifit}(tasklist, \text{cores})$   
**while** cores > 1 **and**  
    $(\sum_i maxwork_i) / (\text{cores} > 1 ? pet^{cores}(C) : C) < \max(1, \rho \cdot \text{cores})$  **do**  
   decrement cores  
    $C \leftarrow \text{multifit}(tasklist, \text{cores})$   
**end while**  
 put the unmerged tasks into bins according to **multifit**  
 update hashmap with all pairs of nodes that are in different bins  
**for all** bins  $b$  **do**  
   **mergeParallel** (tasks in  $b$ )  
**end for**  
 recalc  $drt_{\min}$  for merged predecessors  
 handle overhead :  $est \leftarrow drt_{\min} + \text{cores} > 1 ? pet^{cores}(C) : C$   
 finish time of join node :  $drt = est + work(j)$

---

search to find a schedule for  $n$  independent tasks in  $k \cdot O(n \cdot \log(n))$  with w.c. error bound  $1.22 \cdot opt + \frac{1}{2^k}$ . Experimental results for multifit indicate that the average error is in the order of 1.01 for  $k = 7$ , so slightly above optimal execution times are expected. If better heuristics than multifit are found to solve this specific scheduling problem then these can be plugged in instead. The schedule length

returned by `multifit` must be corrected using the model from Eq. 11 if more than one core is used to obtain realistic data ready times.

`Multifit` is modified to not merge pairs of nodes where it has been already established that merging them is not effective. This information is stored in a hashmap.

The final while loop searches for the biggest number of cores which yields a speedup of at least  $\rho$  per core rather than minimizing the (parallel) execution time of the tasks in question. This avoids that a large number of cores is utilized to achieve small speedups (e.g. 100 cores for speedup 1.01). Obviously, the algorithm could be modified here to minimize the execution times. This might be desirable if the given task graph describes the complete program. Often, hierarchical task graphs are used [10] to represent complex programs so that one individual subgraph describes only part of a larger program that may run in parallel to the subgraph under consideration. In this situation or when energy efficiency is considered, optimizing for speedup per core yields better results as cores are used only if a minimum speedup can be achieved.

Eventually, all tasks scheduled to the same core by `multifit` are merged using **`mergeParallel`** while maintaining that the graph is a Hasse diagram.

Both merge operations update the mapping *um* from unmerged to merged nodes and the DRTs of the merged node (and all nodes reachable from it) by adding the work from all nodes that were merged to the previous DRT. Moreover, both operations update the distances of the nearly shortest paths (NSP), as well as the paths themselves, which represent approximations of the shortest paths inside the merged graph after merging the nodes. They do so by iterating over all ancestors of the merged node, calculating their distances to it, and checking whether there is now a shorter path to a descendant of the merged node over a path that traverses the merged node. In order to retain the Hasse property, transitive edges are removed inside **`mergeParallel`**. Therefore, the NSPs are correct paths, but might not be the actual shortest paths in general.

As a side effect, the algorithm calculates an execution time prediction for the complete task graph in  $O(n^3)$ . If only this prediction is desired the last while loop of **`merge`** in Algorithm 5 and everything beyond that can be omitted.

Our algorithm correctly predicts that the task graph from Fig. 1 will be executed about 1.5 times faster on the specific target platform if all nodes are collapsed into a sequential program compared to the fully parallel program. Of course, the quality of the prediction is highly dependent on the quality of the (target specific) task size estimates. So far, the preconditioning algorithm assumes infinitely many cores. If the maximum parallelism in the result graph does not exceed the available cores of the target hardware then the result graph can be executed without further modifications. This may often be the case, as the preconditioning algorithm removes all inefficient parallelism and the number of available cores in modern hardware increases. If the graph contains too much parallelism after preconditioning, any traditional algorithm may be used to produce a schedule. If this algorithm uses data ready times then it must be modified to incorporate the realistic execution time prediction from Eq. 5 in order to produce realistic schedules.

Running a scheduling overhead corrected version of a traditional scheduling algorithm without the preconditioning algorithm is not sufficient to avoid bad schedules as these algorithms are not aware of the overhead and would treat it like useful computation.

As an alternative, when dealing with finitely many cores, the preconditioning algorithm may be turned into a complete scheduling algorithm. The algorithm is executed  $k$  times in order to find the  $\rho$ -speedup value which produces a task graph with as many or less tasks as the hardware supports.  $\rho$  is obtained by performing a  $k$ -step binary search with  $\rho \in [0 \leq \rho_{\min} \dots 1]$ . This increases the execution time of the algorithm to  $k \cdot O(n^3)$  and finds the optimal  $\rho$  value with an error of  $\frac{1-\rho_{\min}}{2^k}$ .

## 5 Experimental Results

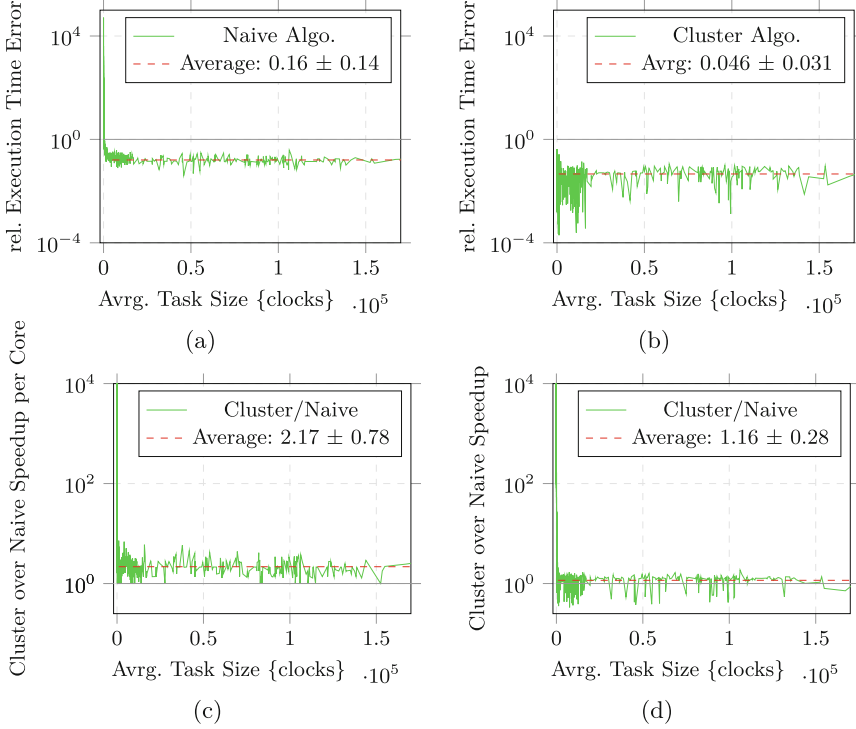
We have generated 1000 random task graphs of varying complexity and a wide range of task sizes using the TGFF library from Dick, Rhodes and Wolf [7] to evaluate the preconditioning algorithm. All task graphs are preconditioned and their execution times averaged 250000 times on the Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz SMP x86\_64 GNU/Linux 3.5.0-37-generic (hyper-threading and speedstep disabled) system. Eight task graphs were removed from the data set because they contained more than 4 parallel tasks after preconditioning and a precise measurement of their execution times was not possible on the 4 core system. The results are presented in Fig. 6.

The relative uncorrected error depicted in Fig. 6(a) shows how much the measured run time of a given task graph deviated from the prediction generated using perfect scheduling (which neglects scheduling overhead). This error is relatively large with an average of  $16\% \pm 14\%$  for average task sizes bigger 5000 clocks (logarithmic scale). For smaller tasks the execution times are mispredicted by up to  $10^7\%$  showing that it is essential to take the overhead into account.

The relative corrected error depicted in Fig. 6(b) shows how much the measured run time of a given task graph deviated from the prediction generated using our algorithm (which incorporates scheduling overhead). The average deviation is  $4.6\%$  with a standard deviation of  $3.1\%$  (for average task sizes bigger 1000 clocks). For extremely small tasks with an average task size near zero the error increases up to  $40\%$  (but is still many orders of magnitude smaller compared to perfect scheduling) for some graphs because the timer resolution on the test system is not good enough to create such small tasks more precisely.

Figure 6(c) shows that the speedup per core achieved by our algorithm relative to perfect scheduling is  $\geq$  one, so that our algorithm never creates a schedule with a speedup per core that is worse than the original schedule. For bigger tasks, the speedup per core is improved on average by  $117\%$ . For smaller tasks the improvement is much stronger because the dynamic scheduling overhead dominates the execution time.

Our algorithms optimizes for speedup per core rather than overall speedup. Nevertheless, Fig. 6(d) shows that on average the overall speedup of the task



**Fig. 6.** Experimental results comparing our clustering algorithm to naive scheduling algorithms that neglect run-time scheduling overhead (perfect scheduling). Averages were calculated for average task sizes bigger 1000 clocks so that they are not biased from the values for on average small task sizes where our algorithm outperforms the classical algorithm by several orders of magnitude.

graphs is improved by 16 % by our algorithm compared to perfect scheduling. Again, for smaller tasks the effect is much more pronounced. Therefore, it can be seen that speedup per core is a measure that does not generally lead to decreased overall speedup. In some specific instances, optimizing for speedup may yield slightly shorter execution times at the expense of utilizing many more cores (and highly reducing energy efficiency).

Our algorithm merges all parallel tasks of local fork/join sub-graphs until the desired speedup per core (and a local speedup  $> 1$ ) is achieved. Globally, a task graph’s critical path consists of sequences of fork/join sub-graphs. Inductively it follows that the task graph’s sink node finishes before or at the same time as in the fully sequentialized version of the task graph so that the global speedup  $\geq 1$ . Also, the speedup per core  $\geq \rho$  as all parallelism that would violate this invariant is removed. This holds if  $\alpha$ ,  $\beta$  and  $\gamma$  from Eq. 11 are chosen so that all task execution times prediction  $\leq$  real execution times. Otherwise, since there is an average error of  $4.6\% \pm 3.1\%$  associated with the predicted task execution times, slight violations of these constraints are possible.



## 6 Related Work

Adve and Vernon [2] predict task graph execution times for a given scheduler model and complete program input data in  $O(n + m)$ . They present a system model where the scheduler and most other parts are modeled using queuing theory. For large task sizes their predictions are fairly good, results for small task sizes are not shown but would suffer from the lack of detailed scheduling overhead modeling. Their results are not applied to scheduling.

McCreary, Khan and Thompson [19] and Kwok [14] compare the makespan and Liu [17] compares worst case bounds of various scheduling heuristics but neglect the real-world execution times and overheads.

Most of the known scheduling heuristics have a complexity of  $O(n^2)$  to  $O(n^3)$  while operating on local information inside the task graph like edge weights and data ready times. Our algorithm is within the usual complexity of  $O(n^3)$  while preprocessing allows us to examine a wider view of the parallelism inside the graph by considering the paths leading from lowest common ancestors via parallel nodes to the next join node (local fork/join sub-graphs).

Liou [16] suggests that clustering before scheduling is beneficial for the final result and McCreary and Gill [18] present a grain packing algorithm. This algorithm is limited to linear and pure fork/join parallelism, more complex graphs are not accounted for in detail and scheduling overhead is neglected. Many other clustering algorithms that do not take scheduling overhead into account have been proposed [15, 22].

Power efficient scheduling has been investigated by [20] and others, taking into account special hardware features to run specific tasks slower and with lower power consumption or better thermal footprint with minimal impact on the makespan. Our algorithm guarantees a minimum core utilization efficiency, so that additional cores are used only if a user defined speedup per core can be achieved. This allows otherwise inefficiently used cores to be turned off completely and can be combined with other power saving techniques.

To the best of our knowledge, none of the previous scheduling algorithms consider scheduling overhead, so in contrast to our algorithm no guarantees on real-world speedup and core utilization efficiency can be given.

## 7 Conclusion and Outlook

We have shown that task graphs which contain tasks with sizes in the order of  $10^5$  clocks and higher are not realistically scheduled by traditional scheduling algorithms as the scheduling overhead is neglected. We derived a sound model for the scheduling overhead of symmetric schedulers and presented a task graph clustering algorithm which unlike previous scheduling algorithms guarantees a real-world speedup and core utilization efficiency. Generally, our algorithm provides a vastly more accurate execution time model compared to existing algorithms and improves the speedup per core in most cases while never making it worse. The effect is viable for large task sizes with improved speedup by 16 %

and improved speedup per core by 117%. For smaller tasks we improve existing methods by several orders of magnitude. Furthermore, we have shown that the scheduling overhead predictions should be incorporated into the existing scheduling algorithms to obtain realistic data ready times.

By extending the scheduler model and preconditioning algorithm to incorporate communication overhead, fully automatic and efficient schedules for cloud and HPC systems may become possible.

## References

1. Adve, V.S., Vernon, M.K.: The influence of random delays on parallel execution times. *SIGMETRICS Perform. Eval. Rev.* **21**(1), 61–73 (1993)
2. Adve, V.S., Vernon, M.K.: Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.* **22**(1), 94–136 (2004)
3. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms* **57**(2), 75–94 (2005)
4. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. *J. ACM* **46**(5), 720–748 (1999)
5. Coffman Jr., E.G., Garey, M.R., Johnson, D.S.: An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.* **7**(1), 1–17 (1978)
6. Darte, A., Robert, Y.P., Vivien, F.: *Scheduling and Automatic Parallelization*. Birkhäuser Boston (2000)
7. Dick, R.P., Rhodes, D.L., Wolf, W.: Tgff: Task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pp. 97–101. IEEE Computer Society (1998)
8. Gerasoulis, A., Yang, T.: On the granularity and clustering of directed acyclic task graphs. *IEEE Trans. Parallel Distrib. Syst.* **4**(6), 686–701 (1993)
9. Girkar, M., Polychronopoulos, C.D.: Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.* **3**(2), 166–178 (1992)
10. Girkar, M., Polychronopoulos, C.D.: The hierarchical task graph as a universal intermediate representation. *Int. J. Parallel Prog.* **22**(5), 519–551 (1994)
11. Graham, R.L.: Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.* **17**(2), 416–429 (1969)
12. Intel. Thread building blocks 4.1 (2013). <http://www.threadingbuildingblocks.org/>
13. Khan, A.A., McCreary, C.L., Gong, Y.: A Numerical Comparative Analysis of Partitioning Heuristics for Scheduling Task Graphs on Multiprocessors. Auburn University, Auburn (1993)
14. Kwok, Y.-K., Ahmad, I.: Benchmarking the Task Graph Scheduling Algorithms, pp. 531–537 (1998)
15. Liou, J.-C., Palis, M.A.: An efficient task clustering heuristic for scheduling dags on multiprocessors. In: *Workshop on Resource Management, Symposium on Parallel and Distributed Processing*, pp. 152–156. Citeseer (1996)
16. Liou, J.-C., Palis, M.A.: A Comparison of General Approaches to Multiprocessor Scheduling, pp. 152–156. IEEE Computer Society, Washington, DC (1997)
17. Liu, Z.: Worst-case analysis of scheduling heuristics of parallel systems. *Parallel Comput.* **24**(5–6), 863–891 (1998)
18. McCreary, C., Gill, H.: Automatic determination of grain size for efficient parallel processing. *Commun. ACM* **32**(9), 1073–1078 (1989)

19. McCreary, C.L., Khan, A., Thompson, J., McArdle, M.: A comparison of heuristics for scheduling dags on multiprocessors. In: Proceedings on the Eighth International Parallel Processing Symposium, pp. 446–451. IEEE Computer Society (1994)
20. Shin, D., Kim, J.: Power-aware Scheduling of Conditional Task Graphs in Real-time Multiprocessor Systems, pp. 408–413. ACM, New York (2003)
21. Indiana University. Open mpi 1(4), 5 (2013). <http://www.open-mpi.org/>
22. Yang, T., Gerasoulis, A.: Dsc: Scheduling parallel tasks on an unbounded number of processors. IEEE Trans. Parallel Distrib. Syst. **5**(9), 951–967 (1994)

Job Scheduling Strategies for Parallel Processing  
18th International Workshop, JSSPP 2014, Phoenix, AZ,  
USA, May 23, 2014. Revised Selected Papers  
Cirne, W.; Desai, N. (Eds.)  
2015, X, 169 p. 60 illus., Softcover  
ISBN: 978-3-319-15788-7