Scheduling Balanced Task Graphs to LogP-Machines

Welf Löwe and Wolf Zimmermann

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, D-76128 Karlsruhe, Germany

Abstract

This article discusses algorithms for scheduling task graphs $G = (V, E, \tau)$ to LogP-machines. These algorithms depend on the granularity of G, i.e. on the ratio of computation $\tau(v)$ and communication times in the LogP cost model, and on the structure of G. We define a class of coarse grained task graphs that can be scheduled with a performance guarantee of $4 \times T_{opt}(G)$, where $T_{opt}(G)$ is the time required for the optimal makespan. Furthermore, we define a class of fine grained task graphs that can be scheduled with a performance guarantee approaching $4 \times T_{opt}(G)$ for increasing problem sizes. The discussed classes of task graphs cover algorithms such as Fast Fourier Transformation, Stencil Computations to solve partial differential equations, matrix multiplication etc.

Key words: Scheduling, LogP-Model, Granularity

1 Introduction

For many parallel programs the communication behavior only depends on the size of the problem and not on the actual input. Using this property for translation and optimization improves the efficiency of the generated code dramatically, cf. the related work Section 2.3. Moreover, programmers may focus on the inherent parallelism of the problems and relax to the properties of the target machine. More specifically, programmers can use a synchronous, shared memory programming model and no data alignment and no mapping of processes onto processors is explicitly required in the source code[1]. Data and processes are distributed automatically.

However, the intension of parallel computing is high performance. Therefore the machine independent programs must be optimized for the respective target

[1] This programming model is equivalent to the PRAM machine model, see [20].

machine. For this optimization, a cost model is required reflecting communication latency, communication overhead, and network bandwidth on the target machines. The LogP-machine [9,10] is a generic machine model, that takes these communication costs into account, cf. Section 2.1. In particular, if the network capacity is exceeded, the transmission of a message may stall.

Since such parallel programs can be represented as task graphs, cf. Section 2.2, our problem is a scheduling problem. The cost function is the running time of the scheduled program on a specific instance of the LogP-model. [22] demonstrates that this scheduling problem is NP-hard even for special cases such as fork- or join-trees of height 1. As discussed in Subsection 2.3, current results on scheduling task-graphs to LogP-machines do not take into account the limited number of available processors. This article discusses approximation algorithms for scheduling special classes of task-graphs to a LogP-machine taking into account the limited number of P processors.

The paper is organized as follows: Section 2 defines the LogP-machine model, task graphs, schedules for the LogP-machine, a notion of granularity γ for general task graphs, and discusses related works. Section 3 defines scheduling algorithms for coarse grained task graphs. All the algorithms have a performance ratio of $1 + 1/\gamma$. Section 3 starts from general task graphs and considers more special classes of task graphs changing the notion of granularity such that with the new notions more task graphs in the class become coarse-grained, i.e. the performace ratio improves for these special classes. Section 4 defines scheduling algorithms for fine grained problems.

2 Foundations

This section defines our the LogP-machine as our machine (and cost) model (cf. Subsection 2.1) and weighted task-graphs our program model (cf. Subsection 2.2). However, programmes would not use task graphs for defining application programs. But, task-graphs correspond to the data-flow of a parallel program on a specific input x. On a synchronous shared memory machine, each layer would be executed in parallel by a synchronous lock-step execution, i.e. after each layer there is a barrier synchronization. When executing a task it first reads its data from the shared memory, then it performs its local computation, and finally it writes its result into the shared memory. If the data-flow via the shared memory depends only on the size of the input but not on the concrete input, the program is called *oblivious.* For oblivious programs we can bridge the gap between parallel programs and task graphs by compilation. Many oblivious parallel programs like FFT, matix and sencil computations compile to balanced task-graphs. Terefore, we discuss this class of task-graphs more detailed.

2.1 LogP-Model

The LogP-model (cf. Fig. 1) assumes a finite number P of processors with local memory. The processors are connected by a data network. It abstracts from the network topology, presuming that the position of the processor in the network has no effect on communication costs. Each processor has its own clock, synchronization and communication are done via message passing. All send and receive operations are initiated by the processor which sends or receives, respectively. From the programmers point of view, the network has no direct connection to the local memory. All communication is done via the processor.

In the LogP-model, communication costs are determined by the parameters L, o, and g. Sending/receiving a message costs time o (*overhead*) on the processor. The time the network connection of this processor is busy with sending/receiving the message into the network is bound by g (*gap*). A processor can not send or receive two messages within time g, but if a processor returns from a send or receive routine, the difference between gap and overhead can be used for computation. The *latency* L is the maximal time between sending a message (completion of send operation) and receiving this message (start of receive operation), provided the message transmission does not stall. There are most $\lceil L/g \rceil$ messages in transit from any or to any processor at any time, otherwise, the message transmission stalls.

If the sending processor is still busy with sending the last bytes of a message while the receiving processor is already busy with receiving, the send and the receive overhead for this message overlap. In this case the latency is negative. This happens on many systems especially for long messages or if the communication protocol is too complicated. L, o, and g have been determined for quite a number of machines; all works confirmed runtime predictions based on the parameters by measurements. In contrast to [1] and [14], we assume the LogP parameters to be constants (as proposed in the introductory LogP papers [9,10]). This assumption is admissible if the message size does not vary in a single program.

An instance of the LogP-machine model with certain parameters L, o, g, and P is denoted a (L, o, g, P)-machine.

2.2 Task Graphs

A *task graph* is a weighted directed acyclic graph (DAG) $G = (V, E, \tau)$ where $\tau : V \to \mathbb{N}$ assigns to each task $v \in V$ a computation time. By predecessor (successor) of a task we always mean a direct predecessor (respectively direct successor). For a task $v \in V$ the set of predecessors (successors) is denoted by $PRED_v$ (respectively $SUCC_v$). A *sink* is a task without successors. $SINKS(G)$ denotes the set of sinks of a task graph G and $\overline{SINKS}(G)$ the set of the non-

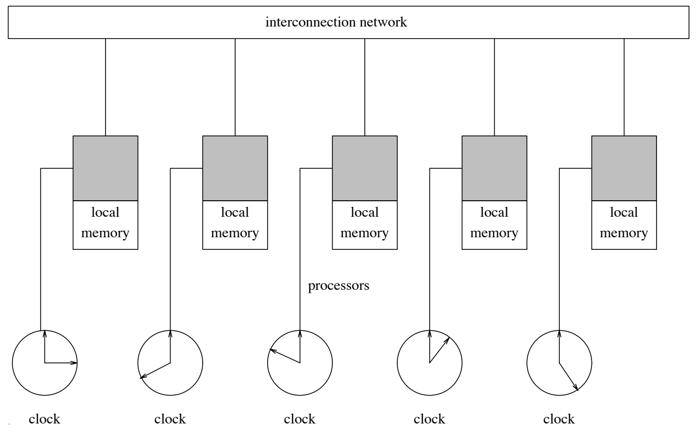

Fig. 1. LogP-machine.

sinks of G. A task v is an *ancestor* of task w iff there is a path from v to w. ANC_w denotes the set of ancestors of task w. G_w denotes the subgraph of G induced by ANC_w. The *height* $h(v)$ *of a task* v *in* G is the length of the longest unweighted path from any task to v in G. The height $h(G)$ of a task graph G is the maximal height of its tasks. It corresponds to the length of its longest unweighted path. The set of all tasks with the same height is called a *layer*. Formally, the ith *layer* of G is $\Lambda_i = \{v \in V \mid h(v) = i\}$, $0 \leq i \leq h(G)$. The maximum cardinality of the layers in G is the width of G and denoted by $P(G)$. A super-layer Λ_i^k is the union of succeeding layers $\Lambda_i \cup \ldots \cup \Lambda_k$.

The *execution time* $T(G)$ of G is the length of the longest weighted path. The *work* $W(G)$ of G is the sum of all task weights. Analogously, the *execution time* $T(\Lambda_i^k)$ *of a super-layer* is the length of the longest weighted path in the subgraph of G induced by Λ_i^k and the *work* $W(\Lambda_i^k)$ *of a superlayer* is the total sum of all weights of the tasks in Λ_i^k.

A task graph G is *balanced* iff tasks in the same layer have same weight, i.e. $h(u) = h(v) \Rightarrow \tau(u) = \tau(v)$, the same outdegree, and the subgraphs induced by the ancestors of the tasks in the same layer are isomorphic (not necessarily identical). Observe that this implies that in-degrees of two tasks are equal if they are in one layer. We therefore also write τ_i, idg_i and odg_i for the computation time, outdegree, and indegree of the tasks in layer Λ_i. A task graph G is *layered* iff there are no transitive edges in G, i.e. for every $(u, v) \in E$, u, v is the only path from u to v.

Remark 1 *Oblivious programs correspond to a family of task graphs G_n where n is the size of the input. For oblivious programs, the execution time $T(G_n)$ and work $W(G_n)$ correspond to the execution time and work of the program, respectively. Furthermore, in order to indicate, that layers etc. may depend on n, we add the parameter n, e.g. $\Lambda_i(n)$ denotes the i-th layer of G_n.*

Task graphs G_n *sufficiently scale* for a (L,o,g,P)-machine iff there are only $O(1)$ layers $\Lambda(n)$ of G_n with $|\Lambda(n)| < P$.

A LogP-schedule for a task graph $G = (V, E, \tau)$ schedules three different kind of operations: it can compute tasks, it can send results of task computations to other processors, and it can receive results of tasks from other processors. It must satisfy the following properties to satisfy the constraints defined by the LogP-machine and the task graph G:

(i) Two operations on the same processor must not overlap in time. Computation of a task v requires time $\tau(v)$. Sending from v or receiving a message sent by v requires time o.
(ii) If a processor computes a task $v \in V$ then any task $w \in PRED_v$ must be computed or received before on the same processor.
(iii) If a processor sends a message of a task to another processor, it must be computed or received before on the sending processor.
(iv) Between two consecutive send (receive) operations there must be at least time g.
(v) For any send operation there is a corresponding receive operation, and vice versa. Between the end of send operation and the beginning of the corresponding receive operation must be at least time L.
(vi) Every $v \in V$ is computed on some processor.

We visualize schedules by Gantt-charts (cf. Figure 6). There is an axis corresponding to processors, and an axis corresponding to time. We place a box at (i, t), if processor i starts at time t an operation. The vertical length of the box corresponds to the cost (i.e. $\tau(v)$ if v is computed and o if it is send or receive operation). Send and receive operations are drawn as black boxes. White boxes denote the computation of a task. The *execution time* of a schedule $T_{\text{sched}}(\mathcal{S})$ is the time when the last operation is finished. $T_{\text{opt}}(G)$ denotes the execution time of an optimal schedule, i.e. the minimal execution time of all schedules for G. A schedule for G is *linear* iff each processor executes a path of G.

We now discuss the communication time required for transmitting messages along an edge (u, v) in the task graph. A sequential execution of send operations of u and receive operations of v leads to an upper time bound $L_{\max}(u, v)$ for communications between u and v of $L + 2o + (odg_u + idg_v - 2)\max(o, g)$ in the worst case, where $odg_u = |SUCC_u|$ and $idg_v = |PRED_v|$. This worst case occurs if message along edge (u, v) is the last message sent by u and the first message received by v. Fig. 2 shows this case. Hence, the upper time bound $L_{\max}(u)$ for communications between non-sink tasks u and its successor tasks on a (L, o, g, P)-machine $L_{\max}(u) = \max\{L_{\max}(u, v) : v \in SUCC_u\}$.

The *granularity of a non-sink task* v relates computation costs to communication costs and is defined as $\gamma(v) = \tau(v)/L_{\max}(v)$. The *granularity of* G is defined as $\gamma(G) = \min\{\gamma(v) : v \in \overline{SINKS}(G)\}$. G is *coarse grained* if $\gamma(G) \geq 1$, otherwise it is called *fine grained.*

The LogP-parameters have been determined for several parallel computers (see Table 1, x denotes the message size in bytes). LogP-based runtime predictions are confirmed for all these machines.

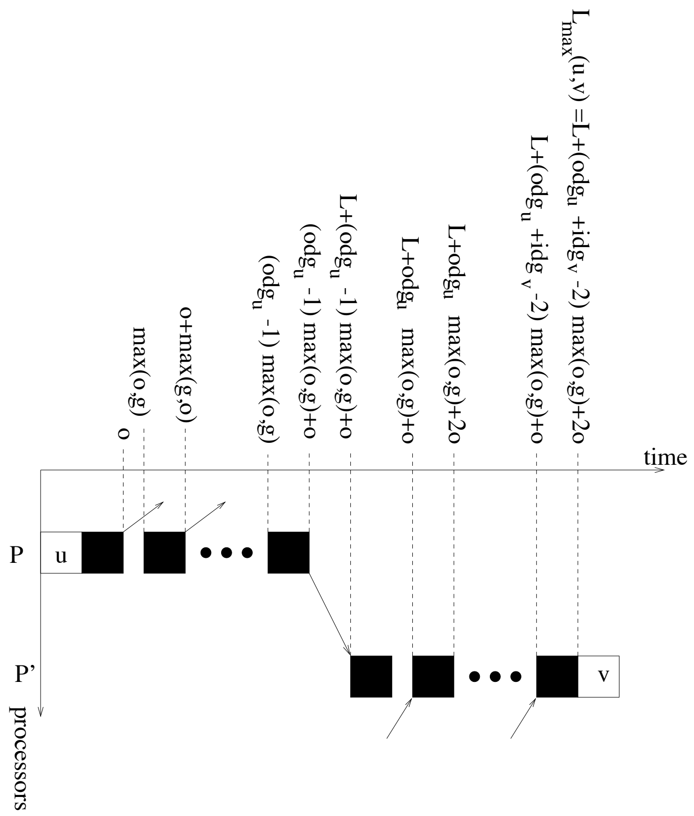

Fig. 2. Upper Time Bound for Communication on the LogP-machine.

Machine	L	o	g	P
CM-5[9,10]	$6\mu s$	$2.2\mu s$	$4\mu s$	512
Parsytec Xplorer [27]	$-21 - 0.82x\ \mu s$	$70 + x\ \mu s$	$115 + 1.43x\ \mu s$	8
ParaStation [27]	$50 - 0.1x\ \mu s$	$3 + 0.112x\ \mu s$	$3 + 0.119x\ \mu s$	4
IBM SP–1 [12]	1000 cycles	8000 cycles	–	
IBM SP–2 [14]	$13 - 0.005x\ \mu s$	$8 + 0.008x\ \mu s$	$10 + 0.01x\ \mu s$	32
Meiko CS-2 [1]	$8.6\mu s$	$1.7\mu s$	$14.2 + 0.03x\ \mu s$	64

Table 1
LogP-Parameters for some Parallel Computers

2.3 Related Work

We first discuss the difficulty of scheduling task graphs to the LogP-machine. Then, we give an overview on scheduling task graphs just considering communication delays. Finally, we discuss the state-of-the art in scheduling task graphs to the LogP-machine.

Papadimitriou and Yannakakis [29] have shown that given a task graph G with unit computation times for each task, communication latency L, and integer $T_{\max}$ it is NP-complete to decide whether G can be scheduled in time $\leq T_{\max}$ (if $P = \infty$). Their results holds no matter whether recomputation of tasks is allowed or not. Finding such schedule remains NP-complete even for simple

DAGs as the concatenation of a join and a fork (when recomputation is not allowed) or fine grained trees (no matter if recomputation is allowed or not) [30]. When recomputation is not allowed, to decide whether a task graph can be scheduled in time ≤ 6 (with $P = \infty$) is NP-complete even if $L = 1$ and each task has computation time one [17]. To decide the same problem with time 5 instead of 6 becomes polynomial time solvable [17]. If recomputation is allowed some coarse grained DAGs (with $\gamma \geq 1$) can also be scheduled optimally in polynomial time [2,11,24].

However, for some special classes of task graphs, such as join, fork, coarse grained (inverse) trees an optimal schedule can be found in polynomial time [6,16,24,36]. In contrast, scheduling those classes of task graphs to LogP-machines remains NP-complete: [28] shows that it is NP-complete to find a restricted schedule (i.e. at least one of the predecessors of task is computed on the same processor) with makespan $\leq T_{\max}$ for coarse grained inverse trees even if $g \leq o$, all tasks have computation time c and assuming constant c, o, and L. [35] shows that the computation of a schedule of length at most B is NP-complete even for fork and join trees and $o = g$.

Much research has been done in recent years on scheduling task graphs with communication delay. Most of these works assume an unrestricted number of processors (see e.g. [16–18,23,29,31,36]). A schedule with performance ratio 2 can be found in polynomial time, if the number of processors is unlimited [29]. [16] shows that linear schedules (i.e. only paths are mapped onto a processor) have a peformance ratio of $1 + 1/\gamma$ where the notion of granularity is a special case of our notion. Scheduling task-graphs with communication delays for a limited number of processors without task duplication is also NP-complete [32]. [32] also discusses an approximation algorithm. [7] discusses these works in more detail. Since [34], the usual way to schedule is to cluster the tasks assuming an unlimited number of processors and then distribute the clusters to the available processors. Sarkar [34] demonstrates by measurements the practical feasability of this approach. However, no guarantee on the performance ratio is given.

The works [3,4,19,22,25,26,28,35,37] investigate scheduling task graphs for the LogP-machine without limitation on the number of processors. [26,37] generalize the result of Gerasoulis and Yang [16] to LogP-machines using linear schedules. [28] shows that optimal linear schedules for trees and tree-like task graphs can be computed in polynomial time if the cost of each task is at least $g - o$. [38] generalizes this result to k-linear schedules. [35] gives some approximation algorithms for join- and fork-trees of height 1. [22] discusses an optimal scheduling algorithm for some special cases of fork graphs. List scheduling under LogP is discussed in [19] (assuming $o = g$). [3,4] consider also list scheduling algorithms and study the practical influence of the overhead, in its spirit similar to [34]. [9,21] investigate task graphs that arise in special applications like various broadcast problems, summation, or FFT. To our knowledge, there is no general approximation scheme with a constant performance ratio, e.g. [25] provides a performance ratio depending on parameters o and g.

Only few works on scheduling task graphs for the LogP-machine consider the bounded number of processors: [13] for coarse-grained task graphs and [27] for fine-grained task graphs. [3] proposes an approach analogous to [34] but does not provide bounds on the performance ratio. The latter just considers special cases. This article is an extension of these results.

To our knowledge, all works except those on scheduling trees, ignore the capacity constraint of the LogP-Machine. In this article we show how the communications can be reorganized such that the capacity constraint is satisfied without increasing the bound on the performance ratio.

3 Coarse Grained Problems

Subsection 3.1 introduces a simple scheduling algorithm using the idea of Brent's Lemma. This algorithm provides efficient schedules for coarse grained programs. However, the resulting schedules are not stall free for every task graph and are poor in performance for fine grained problems. Subsection 3.2 improves the algorithm from the previous section for balanced task graphs in the sense that the resulting schedules are always efficient and stall free for coarse grained problems.

3.1 Brent-Scheduling

Coarse-grained task graphs are sensitive w.r.t. the distribution of the work and relatively robust w.r.t. the distribution of communication. If the width of a task graph is greater than the number of processors and the algorithm is coarse grained, we schedule the computations layer by layer to P processors according to Brent's Lemma [5]. Between the tasks, we insert necessary send and receive operations if required (cf. Algorithm 1).

Theorem 2 *For each task graph $G = (V, E, \tau)$, Algorithm 1 computes a correct schedule on a (L, o, g, P)-machine.*

PROOF. All tasks are scheduled. Because of the layer-by-layer-approach, all precedences in G are guaranteed by the schedule. For each task v, send and receive operations are scheduled such that requirements (ii)-(iv) on LogP-schedules are satisfied, cf. steps (5) to (7). Between succeeding layers L is guaranteed, cf. steps (10) and (11). Hence, each receive operations is scheduled only if the corresponding message is available. ■

Note, that the schedule obtained by Algorithm 1 is not guaranteed to be stall free. If it is stall free, then the performance of the schedules is bounded:

Algorithm 1 *Given a task graph $G = (V, E, \tau)$ and a (L, o, g, P)-machine:*

```
(1)  for all processors $p \in \{0, \dots, P-1\}$ do $ct(p) := 0$;
                         – completion times are initially 0
(2)  for $i := 0$ to $h(G)$ do – consider all layers
(3)     for all $v \in \Lambda_i$ do
(4)        choose a processor $p'$ with $ct(p')$ is minimal;
(5)        schedule sequentially all $|PRED_v|$ receive operations to $p'$
           such that the gap $g$ is guaranteed;
(6)        schedule $v$ to $p'$ immediately after completion of
           the last receive operation;
(7)        schedule sequentially all $|SUCC_v|$ send operations to $p'$
           such that the gap $g$ is guaranteed;
(8)        $ct(p')$ := completion of the last send operation on $p'$;
(9)     endfor
(10)    $ct := \max\{ct(p) + L \mid 0 \leq p < P-1\}$;
(11)    for all processors $p \in \{0, \dots, P-1\}$ do $ct(p) := ct$
(12) endfor
```

Theorem 3 *Let $G = (V, E, \tau)$ be a task graph, $\mathcal{S}$ be its schedule computed by Algorithm 1 for an (L, o, g, P)-machine, and $T_0(G)$ be the time for computing the task graph layer-wise on the $(0, 0, 0, P)$-machine (i.e. tasks of layer Λ_i can be computed only if all tasks of Λ_{i-1} are completed). If $\mathcal{S}$ is stall free, then*

$$T_{\text{sched}}(\mathcal{S}) \leq \left(1 + \frac{1}{\gamma}\right) T_0(G).$$

PROOF. Let $G' = (V', E', \tau')$ be a task graph where $V' = V$, $\tau' = \tau$, and $E' = E \cup E''$ where $(u, v) \in E''$ iff u is computed before v by the same processor P_i of $\mathcal{S}$ and there is no other task that is computed by P_i between u and v. However, we do not account these new edges for the definition of the granularity, because no communication is performed by these edges. $\mathcal{S}$ is a linear schedule for G'. By the results on linear schedules [26], it holds $T_{\text{sched}}(\mathcal{S}) \leq (1 + 1/\gamma(G)) \cdot T(G')$. By construction of G' it holds $T(G') \leq T_0(G)$. Hence, the bound follows. ■

The bound in Theorem 3 is almost tight. The following example demonstrates that it only overestimates the last layer.

Example 4 *Consider the task-graph in Fig. 3 where the computation time of every task is* 1. *It has the granularity* $\gamma = 1/7$ *on the* $(1, 1, 2, 3)$*-machine. Fig. 4 shows a schedule for the task graph obtained by Algorithm 1. It has the processor assignment as shown in Fig. 3. Since, the tasks are equally distributed, it holds* $T_0(G) = 4$. *The execution time of the schedule is* $(1 + 1/\gamma) \cdot 4 = 32$. *The communication phase after level* 3 *is finished at time* $(1 + 1/\gamma) \cdot 3 = 24$.

Remark 5 *The more complicated formula* $T_0(G) + \gamma^{-1} T_0(G')$ *where* $T_0(G')$ *is the maximum completion time of the tasks of layer* $\Lambda_{h(G)-1}$ *is a tight bound.*

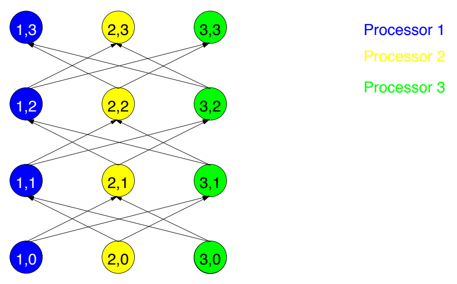

Fig. 3. A Task Graph for Demonstrating Tightness of Bound in Theorem 3

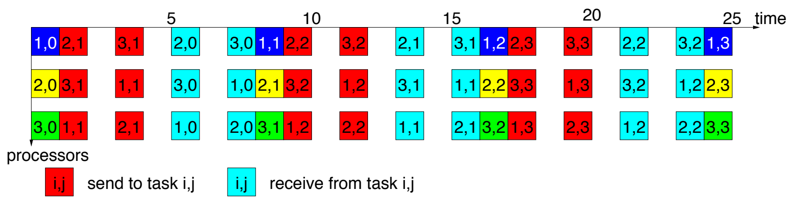

Fig. 4. Schedule for the Task Graph in Fig. 3

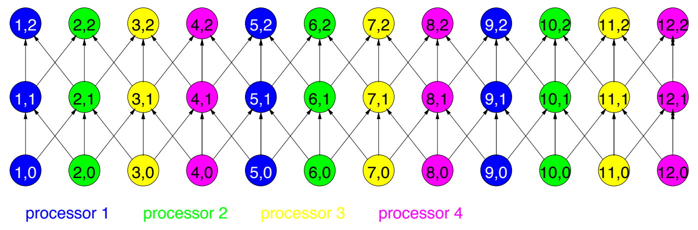

Fig. 5. Task Graph for a One Dimensional Wave Simulation.

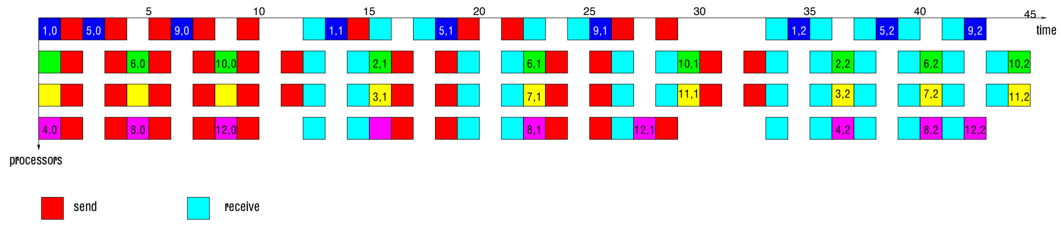

Fig. 6. Schedule of Task Graph from Figure 5 According to Algorithm 1 for the (2, 1, 2, 4)-Machine.

Corollary 6 *Let $G = (V, E, \tau)$ be a task graph, $\mathcal{S}$ be its schedule computed by Algorithm 1 for an (L, o, g, P)-machine. Then:*

$$T_{\text{sched}}(\mathcal{S}) \leq \left(1 + \frac{1}{\gamma}\right) \left(\frac{W(G)}{P} + (1 - 1/P) \sum_{i=0}^{h(G)} T(\Lambda_i)\right).$$

PROOF. According to Brent's Lemma, it holds

$$T_0(G) = \frac{W(G)}{P} + (1 - 1/P) \sum_{i=0}^{h(G)} T(\Lambda_i). \qquad \blacksquare$$

Example 7 *For simulating a one-dimensional wave, a new value for every simulated point is recalculated in every time step according to the current value of this point(y_0), its two neighbors (y_{-1},y_{+1}), and the value of this point one time step before (y'_0). This update is performed by the function*

$$\Phi(y_o, y_{-1}, y_{+1}, y'_o) \; = \; 2 \cdot y_0 \; - \; y'_o \; + \; \Delta_t^2/\Delta_y \cdot 2 \cdot (y_{+1} \; - \; 2 * y_0 \; + \; y_{-1}).$$

The task graph and its schedule for the $(2,1,2,4)$*-machine is shown in Figures 5 and 6, respectively. Its makespan is* 45.

3.2 Balanced Task Graphs

As mentioned before, many parallel programs compile to balanced task-graphs. Therefore, this class of task-graphs is especially interesting from the application point of view. Additionally, we are able to take advantage from their special properties for our scheduling algorithms as the following discussion shows. Alltogether, balanced task-graphs allow for much better performance results than general task-graphs.

For balanced task graphs, the Brent-Scheduling allows for performance bounds in terms of the time for the optimum schedule:

Corollary 8 *For each balanced task graph* $G = (V, E, \tau)$*, Algorithm 1 computes a schedule on a* (L, o, g, P)*-machine and if it is stall free it holds*

$$T_{\text{sched}}(\mathcal{S}) \leq \left(1 + \frac{1}{\gamma}\right)\left(2 - \frac{1}{P}\right) T_{opt}(G). \tag{1}$$

If G *is coarse grained it holds*

$$T_{\text{sched}}(\mathcal{S}) \leq 4 \times T_{opt}(G). \tag{2}$$

PROOF. It holds $T_0(G) \leq W(G)/P + T(G)$. Since both $W(G)/P$ and $T(G)$ are lower bounds of the optimum schedule of G, equation (1) follows directly from Theorem 3. If $\gamma \geq 1$, the first factor is bounded from above by 2 as the second is, hence (2) holds. ■

However, balanced task graphs allow for a better implementation of the communication than the Brent-scheduling (cf. Algorithm 2). Instead of interleaving computation and communication of the tasks of a layer we perform for each layer a receive operation phase, a computation phase and send operation phase. This guarantees that the resulting schedules are stall free. Additionally, this approach allows to pack several small messages with the same destination into one larger message. As the table with the LogP-parameters shows, it is more efficient for many architectures to send such a large message instead of several small messages.

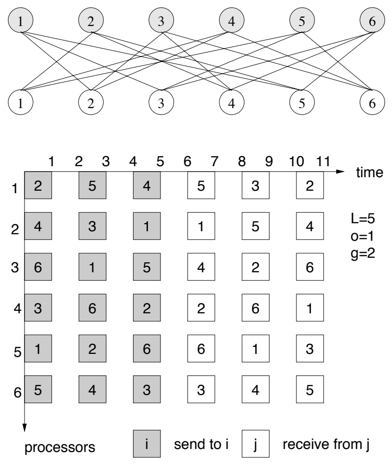

Fig. 7. A 3-relation and its Implementation on LogP

Algorithm 2 *Given a task graph $G = (V, E, \tau)$ and a (L, o, g, P)-machine:*

```
(1)  for $i := 0$ to $h(G)$ do – consider all layers
(2)      partition $\Lambda_i$ into sets $C_p^i$, $p = 0, \dots, P-1$ of almost equal size;
         (i.e. $|\Lambda_i| \bmod P$ sets have size $\lceil |\Lambda_i|/P \rceil$ elements,
         the others have $\lfloor |\Lambda_i|/P$ elements)
(3)  for all processors $p \in \{0, \dots, P-1\}$ do
(4)      schedule the tasks in $C_p^0$ onto processor $p$;
(5)  $ct := \lceil |\Lambda_0| \rceil \cdot \tau_0$;
(6)  for $i := 1$ to $h(G)$ do
(7)      schedule the communications from level $\Lambda_{i-1}$ to higher levels;
           – – to be detailed later
(8)      for all processors $p \in \{0, \dots, P-1\}$ do
(9)          schedule the tasks in $C_p^0$ onto processor $p$;
(10) endfor
```

Since the precedence constraints have to be obeyed, Algorithm 2 is correct if step (7) is implemented correctly. In the following, we assume for simplicity that step (7) implements an exact h-relation, i.e. each processor sends and receives exactly h messages. The goal is to implement the communication "pipelined", as e.g. shown in Figure 7 (the h-relation is visualized as a bipartite graph): all messages sent at time 0 are received at time 6, all messages sent at time 3 are received at time 9, and all messages sent at time 5 are received at time 11.

For implementation of step (7), we construct the (bipartite) *communication graph* $H = (U, V, F)$ where U and V are the set of processors, and $F \subseteq U \times V$ is a multiset. The multiplicity of an edge (u, v) in F equals to the number of messages sent from processor u to processor v (i.e. $|C_u^{(i)} \times C_v^{(i+1)} \cap E|$). It is possible to compute an edge coloring with h colors in time $O(|E| \log(|V| + |U|)$ [8].

The basic idea for construction a pipelined communication is to compute an edge coloring of H, i.e. a mapping $c : F \to \mathbb{N}$ such that $c(e) \neq c(e')$ iff e

and e' are adjacent to a common task. Suppose $|c(F)| = k$, i.e. the coloring requires k colors. Then, the messages are sent in order of the colors of the corresponding edges in H. For the concrete implementation of Step (7), we have to distinguish two cases in order to avoid stalling: $L \geq (h-1)\max(o,g)$ and $L < (h-1)\max(o,g)$.

CASE 1: $L \geq (h-1)\max(o,g)$. Let ct be the completion time of the computation phase of level Λ_i. The pipelining is achieved by scheduling a $send(v)$ on processor u at time $ct + j \cdot \max(o,g)$ and $recv(u)$ on processor v at time $ct + L + o + j \cdot \max(o,g)$ iff $c(u,v) = j$. It holds

Lemma 9 *Suppose that the computation of the tasks in Λ_i is completed at time ct, the following communication phase has to perform a h-relation, and $L \geq (h-1)\max(o,g)$. Then the following communication phase is completed at time $ct + L + 2o + (h-1)\max(o,g)$. Furthermore, the capacity constraints are not violated.*

PROOF. By construction, the last send operations are completed at time $ct+o+(h-1)\max(o,g)$. Since $L \geq (h-1)\max(o,g)$, the first receive operation can be performed at time $L+o$. Hence, the overall communication phase is completed at time $ct+L+2o+(h-1)\max(o,g)$. During this communication phase, at most h messages are in transit from any or two any processor. Since $L \geq (h-1)\max(o,g)$, it holds $h \leq \lceil L/g \rceil$, i.e. no message stalls. ■

Table 2 shows an edge coloring of the bipartite graph in Figure 7. The schedule from Figure 7 is obtained by the approach described in CASE 1.

Color	Edges
0	$(1,2),(2,4),(3,6),(4,3),(5,1),(6,5)$
1	$(1,5),(2,3),(3,1),(4,6),(5,2),(6,4)$
2	$(1,4),(2,1),(3,5),(4,2),(5,6),(6,3)$

Table 2
An Edge Coloring with 3 Colors of the Bipartite Graph in Figure 7

CASE 2: $L < (h-1)\max(o,g)$. The same schedule as in CASE 1 would stall because there might be more than $h > \lceil L/g \rceil$ messages in transit to a processor or from a processor. In order to avoid stalling, every message is received greedily, i.e. as soon as possible after a send operation on the processor is finished at the time when the message arrives. Then, it holds:

Lemma 10 *Suppose that the computation of the tasks in Λ_i is completed at time ct, the following communication phase has to perform a h-relation, and $L < (h-1)\max(o,g)$. Then the following communication phase is completed at time $ct + 2o + 2(h-1)\max(o,g)$. Furthermore, the capacity constraints are not violated.*

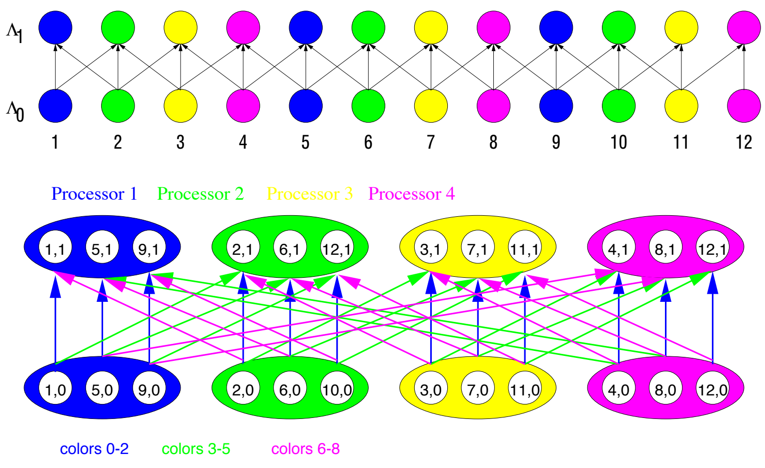

Fig. 8. Bipartite Graph for One Dimensional Wave Simulation.

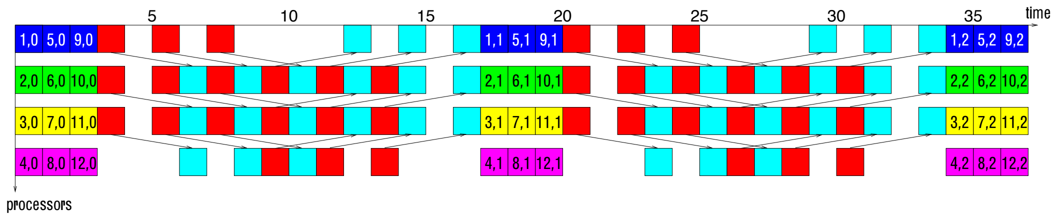

Fig. 9. Schedule for One Dimensional Wave Simulation for the $(2, 1, 2, 4)$-Machine.

PROOF. We prove the bound first in the simplified model without capacity constraints. We use the same approach as in Case 1, i.e. first all send operations are performed, then all receive operations are scheduled. Hence, the first receive operation is scheduled at time $ct + o + (h - 1)\max(o, g)$ (instead of time $o + L$ as in Case 1). Thus, the communication phase is completed at time $ct + 2o + 2(h - 1)\max(o, g)$. Receiving the messages greedily does not increase the idle times. Therefore, the communication phase is also completed at time $ct + 2o + 2(h - 1)\max(o, g)$ when messages are received greedily. Since the messages are received greedily, the capacity constraint is satisfied. ∎

Example 11 *Figure 8 shows the first two levels of the task graph and the bipartite graph obtained for its communication phase. The processor assignment is the result of Step (2) of Algorithm 2. Figure 9 shows the schedule for the* $(2, 1, 2, 4)$*-Machine obtained by Algorithm 2 using the above approaches. The makespan of this schedule is* 37 *in contrast to the makespan* 45 *of the schedule obtained by Algorithm 1.*

It is now possible to define a notion of granularity based on levels instead of tasks. Let $G = (V, E, \tau)$ be a balanced task graph. The *granularity of layer* Λ_i, $0 \leq i < h(G)$ is defined as

$$\hat{\gamma}(\Lambda_i) = \frac{\lceil \Lambda_i / P \rceil \tau_i}{\bar{L}(\lceil \Lambda_i / P \rceil odg_i)},$$

where

$$\bar{L}(h) = \begin{cases} L + 2o + (h-1)\max(o,g) & \text{if } L \geq (h-1)\max(o,g) \\ 2o + 2(h-1)\max(o,g) & \text{if } L < (h-1)\max(o,g) \end{cases}.$$

$\hat{\gamma}(G) = \min\{\hat{\gamma}(\Lambda_i) : 0 \leq i < h(G)\}$ is the *layer-based granularity* of G. With these definitions, it holds:

Theorem 12 *Let $G = (V, E, \tau)$ be a task graph and $\mathcal{S}$ be its schedule for the (L, o, g, P)-machine obtained by Algorithm 2. Then:*

$$T_{\text{sched}}(\mathcal{S}) \leq \left(1 + \frac{1}{\hat{\gamma}(G)}\right)\left(\frac{W(G)}{P} + T(G)\right)$$

PROOF. The makespan is given by

$$\begin{aligned}
T_{\text{sched}}(\mathcal{S}) &\leq \left\lceil \frac{\Lambda_{h(G)}}{P} \right\rceil \tau_{h(G)} + \sum_{i=0}^{h(G)-1} \left\lceil \frac{\Lambda_i}{P} \right\rceil \tau_i \cdot \bar{L}(\left\lceil \frac{\Lambda_i}{P} \right\rceil odg_i) \\
&\leq \left\lceil \frac{\Lambda_{h(G)}}{P} \right\rceil \tau_{h(G)} + \sum_{i=0}^{h(G)-1} \left(1 + \frac{1}{\hat{\gamma}(\Lambda_i)}\right) \left\lceil \frac{\Lambda_i}{P} \right\rceil \tau_i && \text{by definition of } \hat{\gamma}(\Lambda_i) \\
&\leq \left\lceil \frac{\Lambda_{h(G)}}{P} \right\rceil \tau_{h(G)} + \left(1 + \frac{1}{\hat{\gamma}(G)}\right) \sum_{i=0}^{h(G)-1} \left\lceil \frac{\Lambda_i}{P} \right\rceil \tau_i && \text{by definition of } \hat{\gamma}(G) \\
&\leq \left(1 + \frac{1}{\hat{\gamma}(G)}\right) \sum_{i=0}^{h(G)} \left\lceil \frac{\Lambda_i}{P} \right\rceil \tau_i \\
&\leq \left(1 + \frac{1}{\hat{\gamma}(G)}\right) \left(\frac{W(G)}{P} + T(G)\right) && \text{by } \left\lceil \frac{\Lambda_i}{P} \right\rceil \leq \frac{\Lambda_i}{P} + 1 \ \blacksquare
\end{aligned}$$

Although $\gamma(G)$ and $\hat{\gamma}(G)$ are incomparable, Algorithm 2 has several advantages compared to Algorithm 1. First, the schedules are always stall-free. Second, it is often cheaper to send one large message instead of sending many small messages, cf. the extension of LogP-models [4,15]. A slight variation of Algorithm 2 leads to schedules in this extended model.

4 Fine Grained Problems

The basic problem with general scheduling approaches is that they ignore the source program that corresponds to the task graph. Subsection 4.1 shows that the knowledge of the source program can be used to improve the scheduling. For stencil computations, it turns out the performance ratio improves when the problem size increases. The remaining subsections then consider more general task graphs that may also be fine-grained. The basic idea is not to schedule just one layer of the task-graph but to compute super-layers without communications. The consequence is that redundant computations may be introduced. The scheduling algorithm first cluster the tasks, cf. Subsection 4.2, and then schedule the single clusters to processors 4.3. We derive performance ratios for some special cases. One of these special cases are difficult to check at the level of task graphs. We show that the knowledge of the source program corresponding to task graphs provide sufficient conditions.

4.1 Using the Source Program for Scheduling

The purpose of this subsection is to show the benefit of the knowledge of the source program for scheduling. The task-graph of Example 7 belongs to the family of task-graphs corresponding to the following stencil computation:

```
for $t := 0$ to $T$ do
    forall $i = 0$ to $n$ do in parallel
        $a[i] := \Phi(a[i-1], a[i], a[i+1])$
```

It is possible to obtain a much better schedule taking into account this source program. Suppose we have a block-wise distribution of the array a onto the P processors, i.e. array elements $a[0], \ldots, a[n/P-1]$ are stored on processor P_0, $a[n/P], \ldots, a[2n/P-1]$ on processor P_1 etc (cf. Fig. 10). Then there are much less communications required as the schedule in Fig. 11 shows. Thus, using data distribution algorithms in combination with scheduling can lead to better schedules.

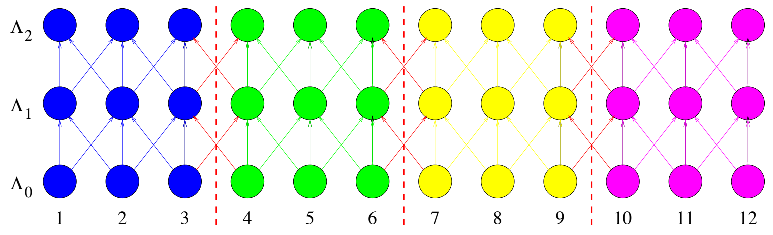

Fig. 10. Partitioning the Task Graph According to Block-Wise Data Distribution

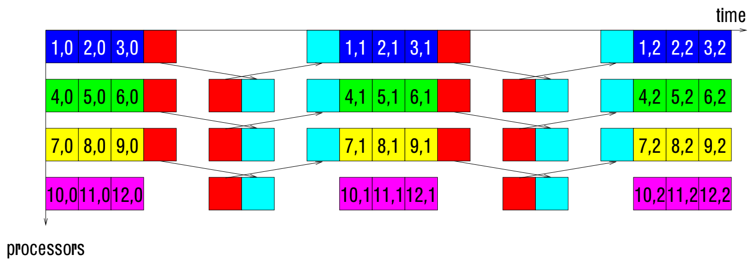

Fig. 11. A Schedule for the $(2,1,2,4)$-Machine based on Block-Wise Data Distribution

In general, we can use the idea of block-wise data distribution for all source programs containing a loop whose body is a parallel loop with the assignment

$$a[j] := \Phi(a[j_1], \ldots, a[j_m])$$

This class of programs are called *one-dimensional stencil computations.* It occurs typically for solving partial differential equations in one spatial dimension. The *locality of the i-th iteration* is defined to be $\delta_i = \max\{|j - j_1|, \ldots, |j - j_m|\}$. Then, it holds

Lemma 13 *Let $\mathcal{P}$ be a one dimensional stencil computation, $G_n = (V, E, \tau)$ be its family of task graphs, and $(\Lambda_0(n), \ldots, \Lambda_{h(G_n)}(n))$ be the layering of G_n. Then, the block-wise data distribution leads to a schedule with at most $2\delta_i$ communications per processor for communicating the results of layer Λ_{i-1} to Λ_i.*

PROOF. We proceed similar as in Algorithm 2. However, instead of greedily assigning the tasks to the processors, we distribute the tasks block-wise: For every level i the task computing array element $a[j]$ is mapped onto the processor storing $a[j]$. The computation of $a[j]$ requires arguments from other processors iff the arguments $a[j_1], \ldots, a[j_k]$ are stored onto different processors. It is easy to see that this can be at most $2\delta_i$ array elements due to blockwise data distribution (at the left and right end of the block). ■

Thus, the bounds of Theorems 3 and 12 can be improved using an improved notion of granularity depending on the locality. Let $G_n = (V_n, E_n, \tau)$ the family of task-graphs corresponding to a one-dimensional stencil computation. The value

$$\bar{\gamma}(\Lambda_i(n)) = \frac{\lceil |\Lambda_i(n)|/P \rceil \tau_i}{(L + 2o + (4\delta_i - 2)\max(o, g))}$$

is the *granularity of layer* $\Lambda_i(n)$, $i > 0$. The *granularity of* G is defined by

$$\bar{\gamma}(G_n) = \min_{i=1}^{h(G_n)} \bar{\gamma}(\Lambda_i(n)).$$

With this new definition of the granularity it holds the

Theorem 14 (Schedules for One-Dimensional Stencil Computations) *Let $\mathcal{P}$ be a one dimensional stencil computation, $G_n = (V_n, E_n, \tau)$ be the family of its task graphs, and $\mathcal{S}_n$ be the family of schedules computed by Algorithm 2 for a (L, o, g, P)-machine. Then*

$$T_{\text{sched}}(\mathcal{S}_n) \leq \left(1 + \frac{1}{\bar{\gamma}(G_n)}\right)\left(\frac{W(G_n)}{P} + T(G_n)\right)$$

PROOF. Lemma 13 implies that the communication phase between layers $\Lambda_{i-1}(n)$ and $\Lambda_i(n)$ costs at most time $L + 2o + (4\delta_i - 2)\max(o, g)$. The remaining of the proof is analogous to the proof of Theorem 12. ■

For small values of δ_i, it holds $\gamma(G_n), \hat{\gamma}(G_n) < \bar{\gamma}(G_n)$. In particular, for many stencil computations, the index accesses are $j_l = j + c_l$ for some constant c_l. Thus, if $n/P \gg c_l$, $l = 1, \ldots, k$ it is $\gamma(G_n), \hat{\gamma}(G_n) \ll \bar{\gamma}(G_n)$, i.e. Theorem 14 gives a much better bound than Theorem 3. However, the improvements are not as good, if $n/P \not\gg c_l$. In the following, we show how to gain back some of the performance by introducing redundant computations.

4.2 Clustering

A *clustering* of G is a finite set $\mathcal{C}(G)$ of subsets of V such that for any $v \in V$ there is an $X \in \mathcal{C}(G)$ with $v \in X$. For clustering we select the super-layers Λ_j^k such that the minimal time for sequential computations necessary to compute a task v in Λ_j^k is at least equal to the time for communicating the results of v to its successors. It is always possible to satisfy this property (at least by computing the whole program sequentially). Algorithm 3 defines such a clustering. These clusters need not to be pairwise disjoint, i.e. the clusterings may introduce redundant computation.

Algorithm 3 *Given a task graph $G = (V, E, \tau)$ and a (L, o, g, P)-machine:*

```
(1) $j := 0$; $k := 0$;
(2) while $k < h(G)$ do
(3)     $k := k + 1$
(4)     if $\bar{L}(\lceil |\Lambda_k|/P \rceil \cdot odg_k) \leq \max\{T_j^k, W_j^k/P\}$ then
(5)         for $v \in \Lambda_k$ do
(6)             $Cl_k(v) := ANC_v \cap \Lambda_j^k$;
                -- $Cl_k(v)$ contains the ancestors of $v$ from a super-layer $\Lambda_j^k$.
(7)         $j := k + 1$;
(8)     endif
(9) endwhile
```

The set of all $Cl(v)$ together defines a clustering $\mathcal{C}(G)$.

The clusters computed by Algorithm 3 have the following structural property:

Lemma 15 *Let $G = (V, E, \tau)$ be a task graph and $\mathcal{C}(G)$ be a clustering computed by Algorithm 3. Then, for every $Cl_k(v) \in \mathcal{C}(G)$, it holds $\Lambda_k \cap Cl_k(v) = \{v\}$ and for every $w \in Cl_k(v)$ there is a path from w to v just containing tasks of $Cl_k(v)$.*

PROOF. If a cluster $Cl_k(v)$ is constructed (line (6)), then v is the only task of layer Λ_k, cf. lines (5)–(6). The second property follows from directly from line (6). ■

Thus, v can be considered as the "root" of cluster $Cl_k(v)$. In a schedule based on the clustering, only the result of task v need to be sent. The following structural property limits the size of clusters:

Lemma 16 *Let $G_n = (V_n, E_n, \tau)$ be a family of layered task graphs corresponding to a program. If $odg_v = O(1)$ for every $v \in V_n$, then all clusters computed by Algorithm 3 have constant height (i.e. only $O(1)$ layers contribute to the cluster).*

PROOF. By steps (4)–(7) of Algorithm 3 and since L, o, g, P, and odg_k are constants and already $W_k^k(n) \geq |\Lambda_k(n)|$, it holds $\bar{L}(\lceil|\Lambda_k|/P\rceil odg_k) = O(\max\{T_j^k, W_j^k/P\})$. Since G_n is layered, only tasks in the last layer of a selected super-layers communicate with succeeding clusters. For each additional layer $\Lambda_i, j \leq i \leq k$ in super-layer Λ_j^k computation time increases by $W_i^i(n)/P$ without affecting communication costs. Hence, there is only a constant number of layers $\Lambda_i, i < k$, required to satisfy the clustering property of our algorithm. ■

4.3 Scheduling

The basic idea is to consider the clusters computed by Algorithm 3 instead of the task graph G and compute the schedule according to Algorithm 2. Basically, it is the schedule of the graph obtained by collapsing into clusters, i.e. the clusters are the new vertices, and there is an edge from cluster $Cl_k(v)$ to $Cl_{k'}(w)$ iff there is a task $u \in Cl_{k'}(w) \setminus Cl_k(v)$ such that $(v, u) \in E$, i.e. in order to compute cluster $Cl_{k'}(w)$ the result of vertex v is required. The weights of the clusters are the sum of the weights of its tasks, respectively. We call this task graph the *clustered task graph of G*.

Lemma 17 *Let G be a balanced and layered task graph, $\mathcal{C}(G)$ be the clustering obtained from Algorithm 3, and H be the clustered task graph of G. Then H is also balanced and layered, the layers of H correspond one-to-one to super-layers of G, and $\tau(Cl_k(v)) \leq \max\{T_j^k, W_j^k/P\}$.*

PROOF. Since G is balanced, for every two clusters $Cl_k(v)$ and $Cl_k(w)$, the subgraphs of G induced by $Cl_k(v)$ and $Cl_k(w)$ are isomorphic. This implies that all clusters on one layer of H have the same weight and the subgraphs of H induced by the ancestors of $Cl_k(v)$ and $Cl_k(w)$ are isomorphic. Hence H is balanced. Lemma 15 directly implies that H is layered. ■

With the above approach, we obtain for balanced and layered inverse trees (or forests) an approximation algorithm with constant performance ratio:

Theorem 18 *Let $G_n = (V_n, E_n, \tau)$ be the family of task graphs corresponding to a program, $Cl(G_n)$ be the clustering obtained by Algorithm 3, H_n be the family of clustered task-graphs of G_n, and $\mathcal{S}_n$ for an $(L, o, g, P)-$machine the schedule for a $(L, o, g, P)-$ machine computed according to Algorithm 2. If the task-graphs G_n are balanced and layered inverse trees, then*

$$T_{\text{sched}}(\mathcal{S}_n) \leq (4 + o(1)) \cdot T_{opt}(G_n).$$

PROOF. The sets of ancestors ANC_v of tasks of the same layer are pairwise disjoint. Hence, no redundancy is introduced and H_n is also an inverse tree. Lemma 17 implies that this tree is balanced and layered. Consider a layer of H_n and suppose it corresponds to super-layer Λ_j^k. Then, since no redundancy is introduced and H_n is balanced and layered, Lemma 15 implies $\tau(Cl_k(v)) = W_j^k(n)/|\Lambda_k|$. Hence, if $P \leq |\Lambda_k|$, it holds $\tau(Cl_k(v)) \leq W_j^k(n)/P$. Thus, $T(H_n) \leq (1+o(1))W(G_n)/P$ since there are only $O(1)$ layers with $P > |\Lambda_k|$ and $\mathcal{C}(G_n)$ is non-redundant.

Clusters are only created if the condition in line (4) of Algorithm 3 is true. Hence, it holds $\bar{\gamma}(H_n) \geq 1$, i.e. H_n is coarse-grained using the layer-based granularity. Thus,

$$\begin{aligned}
T_{\text{sched}}(\mathcal{S}_n) &\leq 2 \cdot (W(H_n)/P + T(H_n)) && \text{by Theorem 12} \\
&= 2 \cdot (W(G_n)/P + T(H_n)) && \text{since } \mathcal{C}(G_n) \text{ is non-redundant} \\
&\leq (4+o(1)) \cdot W(G_n)/P && \text{since } T(H_n) \leq (1+o(1))W(G_n)/P \\
&\leq (4+o(1)) \cdot T_{\text{opt}}(G_n).
\end{aligned}$$

■

The performance guarantee of Theorem 18 depends on the fact that the clustering computed by Algorithm 3 is non-redundant for trees. It can be similarly argued for all balanced and layered task graphs if the clustering is non-redundant. However, it is hard to obtain a structural property different from trees that guarantees that Algorithm 3 computes non-redundant clusterings.

For redundant clusterings, the Algorithm 2 should distribute the clusters such that the redundancy introduced is minimized by scheduling them on the P processors. We formalize this problem as follows: Let $G = (V, E, \tau)$ be a task-graph, and $\mathcal{C}(G)$ be a clustering of G. The *redundancy graph* $G_R = (V_R, E_R, \lambda)$ *of super-layer* Λ_j^k is an edge-weighted undirected graph where $V_R = \{C \cap \Lambda_j^k : C \in \mathcal{C}(G)\}$, $(C', C'') \in E_R$ iff $C' \cap C'' \neq \emptyset$ tasks redundantly, and $\lambda((C', C'')) = \sum_{v \in C' \cup C''} \tau(v)$. The general problem is to find a minimum balanced P-cut[2] in G_R for every super-layer that corresponds to a layer in the clustered task-graph of G. Unfortunately, this problem is NP-hard [33], even for the unbalanced version, and the best approximation for the balanced version is within a factor of $|V_R| \cdot (P-1)/P$ of the optimum. Hence, we cannot expect acceptable solutions for all programs.

Corollary 19 *Let $G_n = (V_n, E_n, \tau)$ be the family of task graphs corresponding to a program, $Cl(G_n)$ be the clustering obtained by Algorithm 3, H_n be the family of clustered task-graphs of G_n. Suppose G_n is layered and balanced. If for almost all n and almost all layers of H_n, the redundancy graph $G_R = (V_R, E_R)$*

[2] A balanced k-cut of a undirected graph is a set of edges that, if removed, divides G into k connected components of equal size. A minimum k-cut of a weighted undirected graph is a k-cut with minimum weight of removed edges.

has $\chi = \omega(1)$ connected components of size $O(P(G_n)/\chi)$, then there is a schedule $\mathcal{S}_n$ for the (L, o, g, P)-machine satisfying the bound of Theorem 18.

PROOF. With the same arguments as Theorem 18, H_n is balanced, layered, and coarse-grained. For all but $O(1)$ layers of H_n, the redundancy graph has at least P connected components. Suppose now, we merge the clusters in the connected components. Then, we obtain another clustering $\mathcal{C}'(G_n)$ of G_n that is non-redundant. Let H_n' be the clustered task-graph obtained by $\mathcal{C}'(G_n)$. H_n' is also layered, balanced, and coarse-grained. Furthermore, the size of its layers correspond to the number of connected components of its redundancy graph G_R. Hence, if G_R for super-layer Λ_j^k has χ connected components, every cluster of H_n' has weight $W_j^k/|\Lambda_j^k|$. We now apply Algorithm 2 is applied on H'. Since there are only $O(1)$ layers of H_n' with size less than P, it holds $T(H_n') \leq (1 + o(1))W(G_n)/P$ with the same arguments as in Theorem 18. Then, the same bound can be derived using the same arguments as in Theorem 18. ■

Remark 20 *The class of tractable programs include now FFT, matrix multiplication (with duplicated second matrix), and sorting networks.*

In the following, we examine classes of source programs that frequently occur in scientific computing. For simplicity, we assume that the source programs $\mathcal{P}$ operate on a single k-dimensional array a, where $n_i, 1 \leq i \leq k$ is the size of the i-th dimension. All statements that are executed in parallel have linear dependencies, i.e $\mathcal{P}$ computes parallel steps of the form

$$a[j_1, j_2, \ldots, j_k] := \Phi(a[j_1^1, j_2^1, \ldots, j_k^1], \ldots, a[j_1^m, j_2^m, \ldots, j_k^m]).$$

The *locality of the $i-th$ iteration step* in $\mathcal{P}$ is defined as:

$$\Delta_i(n_1, \ldots, n_k) = \max_{i=1}^{m} \max_{l=1}^{m} |j_1 - j_1^l|.$$

The *locality of* $\mathcal{P}$ is $\Delta(n_1, \ldots, n_k) = \max_{i=0}^{T(n_1,\ldots,n_k)} \Delta_i(n_1, \ldots, n_k)$ where $T(n_1, \ldots, n_k)$ is the number of parallel steps executed by $\mathcal{P}$. We first consider the one-dimensional case[3]:

Theorem 21 *Let $\mathcal{P}$ be a program on a one-dimensional array that sufficiently scales for a (L,o,g,P)-machine, G_n be the family of task-graphs corresponding to $\mathcal{P}$. program, $\mathcal{C}(G_n)$ be the clustering computed by Algorithm 3, H_n the clustered task-graph of G_n, and $\mathcal{S}_n$ be the schedule for an (L, o, g, P)-machine computed by Algorithm 2 using block-wise distribution of the clusters. If G_n is balanced and layered, and $\Delta(n) = o(P(G_n))$, then*

$$T_{\text{sched}}(\mathcal{S}_n) \leq (4 + o(1)) \cdot T_{opt}(G_n).$$

[3] $n = n_1$ for simplicity.

PROOF. Since G_n is layered, redundancy occurs only between clusters of the same layer, H_n is balanced and layered, and the outdegree of G_n is constant. Consider now a super-layer $\Lambda_j^k(n)$ that corresponds to a layer of H_n. Since G_n is balanced, it holds $\tau(Cl_k(v)) = W_j^k(n)/P$. Hence, since $\mathcal{P}$ sufficiently scales, $T(H_n) = W(G_n)/P + O(1)$ (cf. Theorem 18). Because super-layers are distributed block-wise there are at most $(P-1)\cdot(k-j)\cdot\Delta(n)$ clusters computing tasks redundantly. Thus, the computation of Λ_j^k on P processors using block-wise distribution requires time

$$\begin{aligned}
\mathit{TIME}_j^k &\le W_j^k(n)/P + (k-j)\cdot\Delta(n) & \\
&= W_j^k(n)/P + O(\Delta(n)) & \text{since } k-j = O(1) \text{ by Lemma 16}\\
&= W(\Lambda_j^k(n))/P + o(P(G_n)) & \text{by assumption}\\
&= (1+o(1))W_j^k(n)/P & \text{since } P(G_n) \le W_j^k(n)
\end{aligned}$$

Hence, $W(H_n)/P = (1+o(1))W(G_n)/P$ by adding TIME_j^k for all super-layers. Using similar arguments as in Theorem 18 we conclude

$$\begin{aligned}
T_{\text{sched}}(\mathcal{S}_n) &\le 2\cdot\left(\frac{W(H_n)}{P} + T(H_n)\right) & \text{since } H_n \text{ is coarse-grained}\\
&\le 2\cdot\left(\frac{W(G_n)}{P}(1+o(1)) + T(H_n)\right) & \text{since } \frac{W(H_n)}{P} = (1+o(1))\frac{W(G_n)}{P}\\
&\le 2\cdot\left(\frac{W(G_n)}{P}(2+o(1))\right) & \text{since } T(H_n) = W(G_n)/P + O(1)\\
&\le (4+o(1))T_{\text{opt}}(G_n)
\end{aligned}$$

■

The class of problems we are able to schedule includes now programs for simulations based on the finite-element-method and numerical solutions of partial differential equations in one spatial dimension. We consider now the k-dimensional case:

Corollary 22 *Let $\mathcal{P}$ be a program on a k-dimensional array with dimension that sufficiently scales for a (L,o,g,P)-machine where $n_i, 1 \le i \le k$ is the size of the i-th dimension, $G_{n_1,\dots,n_k}$ be the family of task-graphs corresponding to $\mathcal{P}$. program, $\mathcal{C}(G_{n_1,\dots,n_k})$ be the clustering computed by Algorithm 3, $H_{n_1,\dots,n_k}$ the clustered task-graph of $G_{n_1,\dots,n_k}$, and $\mathcal{S}_{n_1,\dots,n_k}$ be the schedule for an (L,o,g,P)-machine computed by Algorithm 2 using row-major or column-major block-wise distribution of the clusters. If $G_{n_1,\dots,n_k}$ is balanced and layered, and $\Delta(n_1,\dots,n_k) = o(n_j)$ for all $j = 1,\dots,k$, then*

$$T_{\text{sched}}(\mathcal{S}_{n_1,\dots,n_k}) \le (4+o(1))\cdot T_{opt}(G_{n_1,\dots,n_k}).$$

PROOF. W.l.o.g. we assume all k dimension are equal in size, otherwise we extend the dimensions to the size of the largest dimension, say $n_{\max}$. Hence, the overall size of a is $n = n_{\max}^k$. Let a' be the array a linearized in column-major or row-major order. Observe that the task-graph does not change due to this linearization. Hence, we can write $P(G_n)$ instead of $P(n_1, \dots, n_k)$, etc. Array cells in neighbored rows or columns are now stored at a distance of at most $P(G_n)^{\frac{k-1}{k}}$. Therefore the locality becomes now

$$\begin{aligned}
\Delta(n) &\leq P(G_n)^{\frac{k-1}{k}} \cdot \Delta(n^{1/k}, \dots, n^{1/k}) \\
&= P(G_n)^{\frac{k-1}{k}} o(P(G_n)^{1/k}) && \text{by assumption} \\
&= o(P(G_n))
\end{aligned}$$

The claim follows now directly from Theorem 21. ■

Remark 23 *Corollary 22 is constructive. Hence, it defines a sufficient condition for finding a good linearization of the shared data-structure. Additionally, we can easily extend it to the case where we have c of such arrays. In this case we compute the linearization for all arrays separately and interleave the single array cells in a cyclic way, i.e. we take the first cells from the each linearized arrays then the second and so on. This increases the the maximum distances of the single linearizations by c.*

5 Conclusions

In this article, we discussed to different approaches for scheduling families of task-graphs to LogP-machines where these families correspond to oblivious parallel programs. The first approach is based on a general approximation algorithm whose performance ratio depends on the granularity. We showed how the definition of granularity can be changed adding more knowledge about the class of task-graphs and the programs they were derived from in order to obtain a better performance ration. The second approach uses the classical 2 phases of clustering and scheduling. For the LogP-machine, no general constant performance ration is known. This article shows that for special classes of families of task-graphs corresponding to oblivious programs, it is possible to derive a a performance ratio of $4 + o(1)$.

For both approaches, it turned out the source program plays an important role. The first approach uses Brent's Lemma. The performance ratio is $(1 + 1/\gamma)$ where γ is a notion of granularity. Thus, the higher the granularity, the better the schedule. Using the knowlege of the source program leads to better distribution of the tasks to the processors. For stencil computations this lead to a notion of granularity that increases as the problem size increase. Hence almost all task-graphs derived from stencil computations are coarse-grained. A similar observation is made using the second approach: the clustering after the clustering phase is usually redundant. A clever distribution of these clusters to the P processors reduces this redundancy. We showed for a class of source programs that a block-wise distribution of the clusters lead to a performance ratio of $4 + o(1)$.

According to the results of this article, at least the following classes of task graphs/source programs have a good approximation: coarse-grained task-graphs (Theorems 3 and 12), stencil computations (Theorem 14), balanced and layered trees (Theorem 18), FFT, Sorting Networks (Corollary 19), simulations based on finite element methods and numerical solutions of partial differential equations (Theorem 21 and Corollary 22).

The chosen approach shows two different directions for further research: First, it is still an open problem whether there exist a general approximation algorithm with a constant performance ratio for scheduling task-graphs to LogP-machines. It seems that transferring standard approaches from scheduling just considering latencies does not to help to answer this question positive [25]. The second direction is the integration of scheduling approaches into compilers which at least to our knowledge is not yet done. It is a non-trivial task, because it is impossible to explicitly construct the task-graph. Using the source program for computing the schedule might be a first step towards this direction. Another helpful property is that the algorithms presented in this article process the task-graph layer by layer.

Acknowledgements We thank the anonymous referees for the helpful comments. They considerably improved the article. We also thank Alfredo Goldman for reading the paper after the revision.

References

[1] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheimann. LogGP: Incorporating long messages into the LogP-model – one step closer towards a realistic model for parallel computation. In *7th Annual Symposium on Parallel Algorithms and Architectures*, pages 95–105. ACM Press, 1995.

[2] F.D. Anger, J. Hwang, and Y. Chow. Scheduling with sufficient loosely coupled processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[3] M.C.S. Boeres. Versatile Communication Cost Modelling for Multicomputer Task Scheduling Heurstics. PhD Thesis, University of Edinburgh, 1996.

[4] C. Boeres, V.E.F. Rebello, and D. Skillicorn. Static Scheduling Using Task Replication for LogP and BSP-Models. In *Europar'98: Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 337–346, 1998.

[5] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, pages 201 – 206, 1974.

[6] P. Chretienne. A polynomial algorithm to optimally schedule tasks over an ideal distributed system under tree-like precedence constraints. *European Journal of Operations Research*, 2:225–230, 1989.

[7] P. Chretienne and C. Picouleau. Scheduling with Communication Delays. In: P. Chretienne, E.G. Hoffmann, J.K. Lenstra (Eds.) *Scheduling Theory and Its Applications*, pages 65–90, 1995.

[8] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM Journal on Computing*, 11(3):540–546, 1982.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pages 1–12, 1993. published in: SIGPLAN Notices (28) 7.

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[11] S. Darbha and D.P. Agrawal. SBDS: A task duplication based optimal algorithm. In *Scalable High Performance Conference*, 1994.

[12] B. Di Martino and G. Ianello. Parallelization of non-simultaneous iterative methods for systems of linear equations. In *Parallel Processing: CONPAR 94 – VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1994.

[13] J. Eisenbiegler, W. Löwe, and W. Zimmermann. Optimizing parallel programs on machines with expensive communication. In *Europar' 96 Parallel Processing Vol. 2*, volume 1124 of *Lecture Notes in Computer Science*, pages 602–610. Springer, 1996.

[14] Jörn Eisenbiegler, Welf Löwe, and Andreas Wehrenpfennig. On the optimization by redundancy using an extended LogP model. In *International Conference on Advances in Parallel and Distributed Computing (APDC'97)*, pages 149–155. IEEE Computer Society Press, 1997.

[15] J. Eisenbiegler, W. Löwe, and W. Zimmermann BSP, LogP, and Oblivious Algorithms. In *Europar'98: Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 865–874, 1998.

[16] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4:686–701, June 1993.

[17] J.A. Hoogreven, J.K. Lenstra, and B. Veltmann. Three, four, five, six or the complexity of scheduling with communication delays. *Operations Research Letters*, 16:129–137, 1994.

[18] H. Jung, L. M. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of directed acyclic graphs with communication delays. *Information and Computation*, 105:94–104, 1993.

[19] T. Kalinowski, I. Kort, and D. Trystram. List Scheduling of General Task Graphs under LogP-Model. Technical Report 865, Institute of Computer Science, Polish Academy of Sciences, 1998.

[20] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol. A*, pages 871–941. MIT-Press, 1990.

[21] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153. ACM, 1993.

[22] I. Kort and D. Trystram. Scheduling fork graphs under logp with an unbounded number of processors. In *Europar'98: Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 940–943, 1998.

[23] J.K. Lenstra, M. Veldhorst, and B. Veltmann. The complexity of scheduling trees with communication delays. *Journal of Algorithms*, 20:157–173, 1996.

[24] W. Löwe and W. Zimmermann. On finding optimal clusterings in task graphs. In N. Mirenkov, editor, *Parallel Algorithms/Architecture Synthesis pAs'95*, pages 241–247. IEEE, 1995.

[25] W. Löwe and W. Zimmermann. Upper time bounds for executing pram-programs on the logp-machine. In M. Wolfe, editor, *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 41–50. ACM, 1995.

[26] Welf Löwe, Wolf Zimmermann, and Jörn Eisenbiegler. On linear schedules for task graphs for generalized logp-machines. In *Europar'97: Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 895–904, 1997.

[27] W. Löwe, J. Eisenbiegler, and W. Zimmermann. Optimizing parallel programs on machines with fast communication. In *9. International Conference on Parallel and Distributed Computing Systems*, pages 100–103, 1996.

[28] M. Middendorf, W. Löwe, and W. Zimmermann. Scheduling inverse trees under the communication model of the logp-machine. *Theoretical Computer Science* 125(3):137–168, 1999.

[29] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322 – 328, 1990.

[30] C. Picouleau. New complexity results on the UET-UCT scheduling algorithm. In *Proc. Summer School on Scheduling Theory and its Applications*, pages 187–201, 1992.

[31] J. Siddhiwala and L.-F. Cha. Path-based task replication for scheduling with communication cost. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 186–190, 1995.

[32] V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. In *Discrete Applied Mathematics* 18:55 – 71, 1987.

[33] H. Saranand V. Vazirani. Finding k-cuts within twice the optimal. In *32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 743–751. IEEE Computer Society, 1991.

[34] V. Sarkar. Partitioning and Scheduling Parallel Programs for Multiprocessors. Research Monographs in Parallel and Distributed Computing, MIT-Press, 1989.

[35] J. Verriet. Scheduling tree-structured programs in the LogP-model. Technical Report UU-CS-1997-18, Dept. of Computer Science, Utrecht University, 1997.

[36] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[37] W. Zimmermann and W. Löwe. An approach to machine-independent parallel programming. In *Parallel Processing: CONPAR 94 – VAPP VI*, volume 854 of *Lecture Notes in Computer Science*, pages 277–288. Springer, 1994.

[38] Wolf Zimmermann, Martin Middendorf, and Welf Löwe. On optimal k-linear scheduling of tree-like task graphs for logp-machines. In *Europar'98: Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 328–336, 1998.

A Summary of Notations

$G = (V, E, \tau)$	task graphs
$PRED_u, SUCC_u$	direct predecessors/successors of task u
ANC_u	ancestors of task u
$SINKS(G), \overline{SINKS}(G)$	sinks/non-sinks of G
$h(G)$	length of longest unweighted path of G
$T(G)$	length of longest weighted path of G
$W(G)$	sum of all weights of tasks of G
Λ_i	layer i of task-graph G
Λ_j^k	super-layer, i.e. $\Lambda_j \cup \cdots \cup \Lambda_k$
$W_j^k(G)$	sum of all weights of tasks of super-layer Λ_j^k
$P(G)$	width of G, i.e. the maximum size of the layers of G
$\gamma(G), \hat{\gamma}(G), \bar{\gamma}(G)$	different notions of granularity
$T_{\text{sched}}(\mathcal{S})$	makespan of schedule $\mathcal{S}$
$T_{\text{opt}}(G)$	minimal makespan of schedules for G
$\bar{L}(h)$	time for scheduling a h-relation on LogP
δ_i, Δ	different notions of localities of source programs