ELSEVIER

Theoretical Computer Science 268 (2001) 67–90

Theoretical Computer Science

www.elsevier.com/locate/tcs

On-line scheduling of parallel jobs with runtime restrictions

Stefan Bischof[*,1], Ernst W. Mayr

SSA SoftSolutions GmbH, Unterkreuthweg 3, D-86444 Mühlhausen, Germany

Accepted May 2000

Abstract

Consider the execution of a parallel application that dynamically generates parallel jobs with specified resource requirements during its execution. We assume that there is not sufficient knowledge about the running times and the number of jobs generated in order to precompute a schedule for such applications. Rather, the scheduling decisions have to be made on-line during runtime based on incomplete information. We present several on-line scheduling algorithms for various interconnection topologies that use some a priori information about the job running times or guarantee a good competitive ratio that depends on the runtime ratio of all generated jobs. All algorithms presented have optimal competitive ratio up to small additive constants, and are easy to implement. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: On-line algorithm; Scheduling; Parallel job; Makespan; UET

1. Introduction

The efficient operation of parallel computing systems requires the best-possible use of the resources that a system provides. In order to achieve an effective utilization of a parallel machine a smart coordination of the resource demands of all currently operating applications is necessary. Consequently, the task of a scheduler is to cleverly assign the resources, most prominently the processors, to the jobs being processed. For the case of sequential jobs, i.e., jobs that require exactly one processor for execution, the involved scheduling problems have been studied intensively for decades [4]. But in many situations the problem arises to find a schedule for a set of parallel jobs [10–12, 4]. Such a set could, for example, be a parallel query execution plan generated by the query optimizer of a parallel database management system [23, 18].

The model studied in this paper assumes that each parallel job demands a fixed number of processors or a specified sub-system of a certain size and topology (depending

* Corresponding author.

E-mail addresses: stefan@bischof-web.de (S. Bischof), mayr@in.tum.de (E.W. Mayr).

[1] http://wwwmayr.in.tum.de/

on the underlying structure of the parallel machine considered) for its execution. It is not possible to run a parallel job on fewer processors than requested, and additional processors will not decrease the running time. This reflects the common practice that the decision on the number of processors is made before a job is passed to the scheduler based on other resource requirements like memory, disk-space, or communication intensity. The processors must be allocated exclusively to a job throughout its execution, and a job cannot be preempted or restarted later. This is a reasonable assumption because of the large overhead for these activities on parallel machines. Furthermore, there may be precedence constraints between the jobs. A job can only be executed if all of its predecessors have already completed execution. Most frequently, precedence constraints arise from data dependencies such that a job needs the complete input produced by other jobs before it can start computation.

We are concerned with on-line scheduling throughout this paper to capture the fact that complete a priori information about a job system is rarely available. However, it has been shown [13, 25] that the worst-case performance of any deterministic or randomized on-line algorithm for scheduling parallel job systems with precedence constraints and arbitrary running times of the jobs is rather dismal, even if the precedence constraints between the jobs are known in advance. Therefore, we study the case that there is some a priori knowledge about the execution times of the individual jobs but the dependencies are unknown to the scheduler.

Three different gradations for this additional knowledge are studied in this paper. The first model of runtime restrictions requires that all job running times are equal and that this fact is known to the on-line scheduler. We give a level-oriented on-line algorithm for this problem that repeatedly schedules a set of available jobs using Bin Packing and collects all jobs that arrive during a phase for execution in the next phase. We show that this algorithm is 2.7-competitive if the First Fit heuristic is used. Due to a lower bound of 2.691 for every deterministic on-line scheduler, our algorithm is almost optimal. Our algorithm can be used for parallel systems that support arbitrary allocation of processors to jobs and one-dimensional arrays. For hypercube connected machines, we present a very similar, optimal on-line scheduling algorithm with competitive ratio 2.

We then explore the entire bandwidth between unit and arbitrary execution times and capture the variation of the individual job running times by a characteristic parameter that we call *runtime ratio* (the quotient of the longest and shortest running time). The results for the proposed on-line schedulers demonstrate a smooth, linear transition of the competitive ratio from the case of unit execution times to unrelated execution times that is governed by the runtime ratio. Our second model postulates that the runtime ratio of a job system is reasonably small and that the on-line scheduler knows the shortest execution time (but not the runtime ratio itself). A family of job systems with runtime ratio $T_R \geqslant 2$ is given that bounds the competitive ratio of any deterministic on-line scheduler by $(T_R+1)/2$ from below. We note that the structure of the dependency graph is an out-forest in all of our lower bound proofs. This bound remains valid even if the scheduler knows the actual runtime ratio in advance. An on-line scheduler designated restricted runtime ratio (RRR) for parallel systems supporting arbitrary allocations is described, and we demonstrate a competitive ratio of $T_R/2+4$ for this algorithm for any job system with runtime ratio $\leqslant T_R$. Therefore, the RRR algorithm is nearly optimal up to a small additive constant. The assumption that the shortest execution time is known to the on-line scheduler can be dropped without much loss of competitive performance. We present a modified algorithm called RRR_ADAPTIVE for this third model, and show it to be $T_R/2+5.5$ competitive.

The remainder of this paper is organized as follows. In Section 2 we introduce our scheduling model, some notation and definitions, as well as basic techniques for analyzing on-line scheduling algorithms.We then discuss previous and related work on on-line scheduling of parallel jobs in Section 3. Section 4 presents nearly optimal on-line schedulers for jobs with unit execution time, whereas in Section 5 we study job systems where the ratio of the running times of the longest and shortest job is bounded. Again, we describe and analyze on-line scheduling algorithms that are optimal up to small additive constants. Finally, we present our conclusions in Section 6.

2. Preliminaries

Let N denote the number of processors of the parallel computer system at hand. A (*parallel*) *job system* is a non-empty set of jobs $\mathscr{J}=\{J_1,J_2,\ldots,J_m\}$ where each job specifies the type and size of the sub-system that is necessary for its execution together with precedence constraints among the jobs in $\mathscr{J}$ given as a partial order $\prec$ on $\mathscr{J}$. If $J_a \prec J_b$, J_b cannot be scheduled for execution before J_a is completed. A *task* is a job that requires one processor for execution, and a job system that only contains tasks is a *sequential* job system.

A *schedule* for a job system $(\mathscr{J},\prec)$ is an assignment of the jobs to processors and start-times such that:

- each job is executed on a sub-system of appropriate type and size,
- all precedence constraints are obeyed,
- each processor executes at most one job at any time,
- jobs are executed non-preemptively and without restarts.

The interconnection topology of the parallel computer system may impose serious restrictions on the *job types* that can be executed efficiently on a particular machine. On a hypercube, for example, it is reasonable to execute jobs only on subcubes of a certain dimension rather than on an arbitrary subset of the processors. On the other hand, a number of interconnection networks do not restrict the allocation of processors to parallel jobs. For example, the CLOS-network of the very popular *IBM RS/6000SP* system, which uses an oblivious buffered wormhole routing strategy, justifies the assumption that the running time of a job only weakly depends on a specific processor allocation pattern (see [1, p. 512] for a short description of this system and [28] for in-depth information on its interconnection network). Therefore, the various types of interconnection networks have to be treated separately.

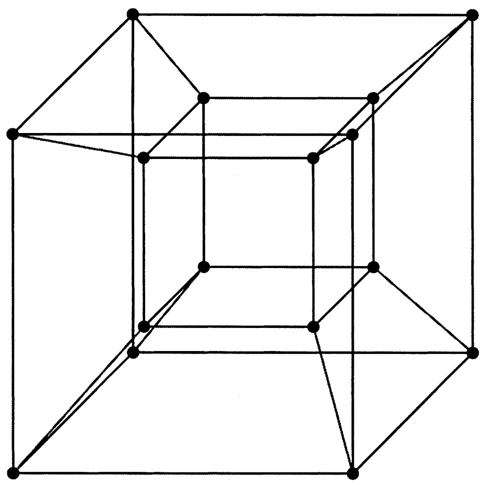
Fig. 1. Four-dimensional hypercube.

The *complete model* assumes that a job J_a requests n_a processors ($1 \leqslant n_a \leqslant N$) for execution and any subset of processors of size n_a may be allocated. The terminology has been chosen in analogy to a complete graph on N nodes. The *r-dimensional hypercube* (see Fig. 1) consists of $N = 2^r$ processors, labeled from 0 to $N - 1$, and has $r2^{r-1}$ point-to-point communication links. Two processors are connected iff the binary representations of their labels (an r-bit string) differ in exactly one bit. As a consequence, each processor is directly connected to $r = \log_2 N$ other processors (see [22] for properties of hypercubes). A job J_a can only request a d_a-dimensional subcube ($0 \leqslant d_a \leqslant r$) for its execution.

Another topology frequently used for parallel computing is the *r-dimensional array* with side-lengths $(N_1, N_2, \ldots, N_r)$, $N_i \geqslant 2$ for $i = 1, 2, \ldots, r$ (also called r-dimensional grid or mesh). The label of a processor is an r-dimensional vector $x = (x_1, x_2, \ldots, x_r)$ with $0 \leqslant x_i < N_i$ for $i = 1, 2, \ldots, r$. Two processors x and y are connected iff $\sum_{i=1}^{r} |x_i - y_i| = 1$. Note that hypercubes form the subclass of arrays with side-length 2 in every dimension. Eligible job types are sub-arrays with side-lengths $(N_1', N_2', \ldots, N_r')$, $1 \leqslant N_i' \leqslant N_i$. The dimension of a job can be less than r if one or more of the N_i' are equal to 1.

It is always possible to transform a job system $(\mathscr{J}, \prec)$ into a directed acyclic graph $D = (\mathscr{J}, E)$ with $(J_a, J_b) \in E \Leftrightarrow J_a \prec J_b$. Removing all transitive edges from D we obtain the *dependency graph* induced by $(\mathscr{J}, \prec)$ (see Fig. 4 for an example). We call two jobs J_a and J_b *dependent* if $J_a \prec J_b$ or $J_b \prec J_a$, and *independent* otherwise. We shall use the terms *dependency* and *precedence-constraint* interchangeably in this paper. The *length of a path* in the dependency graph induced by $(\mathscr{J}, \prec)$ is defined as the sum of the running times of the jobs along this path. A path is called *critical* if its length is maximum among all paths in the dependency graph induced by $(\mathscr{J}, \prec)$.

Assume that all jobs have running time 1 and let P be a longest path in D ending at job J. Then *depth*(J) is defined as the number of nodes in P. If we partition a

Table 1
Frequently used notations

T_{opt}	Length of an optimal off-line schedule for $(\mathscr{J}, \prec)$
T_{ALG}	Length of a schedule for $(\mathscr{J}, \prec)$ generated by Algorithm ALG
$T_{\max}$	Maximal length of any path in the dependency graph induced by $(\mathscr{J}, \prec)$
$t_{\min}$	Minimal running time of any job in $\mathscr{J}$
$t_{\max}$	Maximal running time of any job in $\mathscr{J}$
$\|S\|$	Length of a schedule S
$T_{<\alpha}$	Total time of a schedule for $(\mathscr{J}, \prec)$ when the efficiency is less then α, $0 \leqslant \alpha \leqslant 1$

schedule for such a unit execution time (*UET*) job system into timesteps of length 1, the depth of a job indicates the earliest possible timestep (*EPT*) J can be scheduled. The ith *level* of D is the set of (independent) jobs $\{J \in \mathscr{J} \mid \text{depth}(J) = i\}$. Motivated by the above observation, a level of D is often referred to as an EPT level of the corresponding job system $(\mathscr{J}, \prec)$.

A job is *available* if all predecessors of this job have completed execution. An on-line scheduling algorithm is only aware of available jobs and has no knowledge about their successors. We assume that the on-line scheduler receives knowledge about a job as soon as the job becomes available. This event, however, may depend on earlier scheduling decisions.

The *work* of a job is defined as the number of requested processors, multiplied by its running time. A schedule preserves the work of a job if the processor-time product for this job is equal to its work. The *efficiency* of a schedule at any time t is the number of busy processors at time t divided by N. In general, the running time of a job is also unknown to the on-line scheduler and can only be determined by executing a job and measuring the time until its completion. In Section 4, though, we study the case of unit execution times and therefore restrict the on-line model there to the case of unknown precedence-constraints.

Throughout the paper we use the notations shown in Table 1 (cf. [25, 13]) for a given job system $(\mathscr{J}, \prec)$. To simplify our presentation, we do not attach the job system or schedule as arguments to the notations in Table 1. The relationships should always be clear from the context. Further notation is introduced when needed.

Our goal is to generate schedules with *minimum makespan*, i.e. to minimize the completion time of the job finishing last. We evaluate the performance of our on-line scheduling algorithms by means of competitive analysis [27, 5, 15]. A deterministic on-line algorithm ALG is called *c-competitive* if $T_{\text{ALG}} \leqslant cT_{\text{opt}}$ for *all* job systems and *arbitrary* N. The infimum of the values $c \in [1, \infty]$ for which this inequality holds is called the *competitive ratio* of ALG. The competitive ratio clearly is a worst-case measure. It is intended to compare the performance of different on-line algorithms that solve the same problem, since it is in general impossible to compute an optimal solution without complete knowledge of the problem instance. An *optimal* on-line algorithm is one with a best-possible competitive ratio. See [5, 15] for a thorough treatment of the fundamental concepts in on-line computation and competitive analysis.

The following two lemmata provide useful tools for the competitive analysis of our scheduling algorithms.

Lemma 1. *Let S be a schedule for a job system $(\mathcal{J}, \prec)$ such that the work of each job is preserved. Let $0 \leqslant \alpha_1 \leqslant \alpha_2 \leqslant 1$ and $\beta \geqslant 0$. Suppose that the efficiency of S is at least α_1 at all times and $T_{<\alpha_2} \leqslant \beta T_{\text{opt}}$. Then*

$$|S| \leqslant \left(\beta + \frac{1-\alpha_1\beta}{\alpha_2}\right) T_{\text{opt}}.$$

See [25] for a proof of this lemma.

Lemma 2. *Consider a schedule for a job system $(\mathcal{J}, \prec)$. Then there exists a path of jobs in the dependency graph induced by $(\mathcal{J}, \prec)$ such that whenever there is no job available to be scheduled, some job of that path is running.*

This lemma is due to Graham [19, 20]. The proof given there still holds for parallel jobs since it uses only the structure of the dependency graph.

3. Previous and related work

Extensive work on non-preemptive on-line scheduling of parallel jobs with or without precedence-constraints was done by Feldmann et al. [13, 25, 14]. However, these results for general parallel job systems are bad news for users of parallel computers since they show that no deterministic on-line scheduler for N processors can have competitive ratio better than N. That is, the competitive ratio is asymptotically unbounded, and even randomization cannot improve this unsatisfactory situation substantially.

One possibility to improve the performance is to restrict the maximum job size to λN processors, $0 < \lambda < 1$. Given this restriction it has been shown that the GREEDY algorithm is optimal for the complete model with competitive ratio $1 + 1/(1-\lambda)$. Setting $\lambda = 1/2$, for example, yields a 3-competitive algorithm. Another alternative is the use of *virtualization*. This means that a parallel job J_a which requests n_a processors is executed on a smaller number of processors n'_a by the use of simulation techniques with a predetermined increase in running time. Under the assumption of proportional slowdown (the running time of a job is enlarged by the factor n_a/n'_a) it can be shown that there is an optimal on-line scheduler for the complete model with competitive ratio $1 + \Phi$, where $\Phi = (1 + \sqrt{5})/2$ is the golden ratio. This improves a previous off-line result of Wang and Cheng [29] with asymptotic performance guarantee 3. For the hypercube, an $O(\log N / \log\log N)$-competitive algorithm has been given, and similar results [13, 25] hold for arrays. The two approaches just described can be combined to yield an optimal on-line scheduler with competitive ratio $2 + (\sqrt{4\lambda^2 + 1} - 1)/2\lambda$ for the complete model.

Both approaches, though, have a severe drawback that arises due to the memory requirements of parallel jobs. Restricting the maximum size of a job to λN processors can thus severely restrict the problem size that can be solved on a particular machine. This is often unacceptable in practice because solving large problems is the main reason for the use of parallel computers besides solving problems fast. Virtualization may be impossible or prohibitively expensive if such memory limitations exist.

The job systems used in the lower bound proofs in [13, 25] for the general case reveal an unbounded ratio of the running times of the longest and shortest job. Therefore, we think it necessary to study the influence of the individual running times on the competitive ratio of on-line schedulers for our scheduling problem. To gain insight into this relationship it is only natural to start with unit execution times as is done in Section 4.

To fill the gap between these two extremes – totally unrelated running times versus unit execution times – we identify the runtime ratio (the ratio of the running time of the longest and shortest job) as the distinctive parameter of a job system for the achievable competitive ratio. The importance of this parameter has also been demonstrated recently in [6] for *off-line* scheduling of jobs with multiple resource demands, both malleable (allow for virtualization with proportional slowdown) and non-malleable.

Although we are interested in on-line scheduling, it might be appropriate to briefly mention some complexity results for the corresponding off-line problems. Not surprisingly, almost any variant of these scheduling problems is $\mathcal{NP}$-hard. Błażewicz et al. [3] have proved that it is strongly $\mathcal{NP}$-hard to compute optimal schedules for parallel job systems with unit execution times and no dependencies if N is part of the problem instance. For any fixed N they showed that the problem can be solved in polynomial time. Furthermore, it is known [17] that the problem is $\mathcal{NP}$-hard for sequential job systems with precedence constraints that are the disjoint union of an in-forest and and an out-forest. The scheduling problem for parallel job systems with arbitrary job running times and without dependencies is strongly $\mathcal{NP}$-hard for every fixed $N \geqslant 5$ [8]. If precedence constraints consisting of a set of chains are involved, the problem of computing an optimal 2-processor schedule for a parallel job system is also strongly $\mathcal{NP}$-hard [8].

4. Jobs with unit execution time

In this section, we restrict our model to the case where all jobs have the same execution time. When the dependency graph is known to the scheduler this problem has been intensively studied by Garey et al. [16]. We show that similar results hold in an on-line environment, where a job is available only if all its predecessors have completed execution.

4.1. Complete model

The Level algorithm (see Fig. 2) collects all jobs that are available from the beginning. Since available jobs are independent we can easily transform the problem

```
Algorithm LEVEL(PACK):
begin
    while not all jobs are finished do
        begin
            $\mathscr{A} := \{J \in \mathscr{J} \mid J$ is available$\}$;    // next EPT level
            schedule all jobs in $\mathscr{A}$ according to PACK;
            wait until all scheduled jobs are finished;
        end;
end.
```

Fig. 2. The LEVEL(PACK) algorithm.

of scheduling these jobs to the BIN PACKING problem: the size of a job divided by N is just the size of an item to be packed, and the time-steps of the schedule correspond to the bins (see [7] for a survey on BIN PACKING). Let PACK be an arbitrary BIN PACKING heuristic. We parameterize the LEVEL algorithm with PACK to express the fact that a schedule for a set of independent jobs is generated according to PACK. Thereafter, the available jobs are executed as given by this schedule. Any jobs that become available during this execution phase are collected by the algorithm. After the termination of all jobs of the first level a new schedule for all available jobs is computed and executed. This process repeats until there are no more jobs to be scheduled.

First, we use the Next-Fit (NF) bin-packing heuristic for scheduling on each level. NF packs the items in given order into a so-called active bin. If an item does not fit into the active bin, the active bin is closed and never used again. A previously empty bin is opened and becomes the next active bin.

Theorem 3. LEVEL(NF) *is* 3-*competitive.*

Proof. The number of iterations of the while-loop is exactly the length of a critical path in the dependency graph. There are two possibilities for each level:

1. The partial schedule for this level has length 1. Let T_1 denote the number of levels of this type.
2. The partial schedule for this level has length $\geqslant 2$. By the packing rule of NF it is clear that the average efficiency of 2 consecutive time-steps in such a partial schedule is $> 1/2$. From this we conclude that the average efficiency of all time-steps but maybe the last one is $> 1/2$. Let T_2 denote the number of final time-steps with efficiency $< 1/2$ in partial schedules for levels of this type.

Since $T_1 + T_2 \leqslant T_{\max} \leqslant T_{\text{opt}}$ we can apply Lemma 1 with $\alpha_1 = 1/N$, $\alpha_2 = 1/2$, $\beta = 1$, yielding:

$$T_{\text{LEVEL(NF)}} \leqslant \left(3 - \frac{2}{N}\right) T_{\text{opt}}. \qquad \square$$

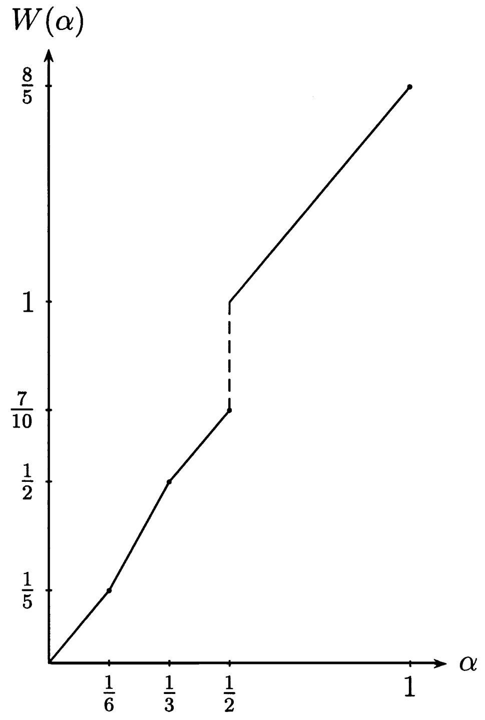

Fig. 3. Weighting function for the analysis of Bin Packing.

Since NF can be implemented to run in linear time (in the number of items to be packed) the scheduling overhead is very low when NF is used to compute partial schedules. Now, we use the first-fit (FF) bin-packing heuristic instead of NF to achieve a better competitive ratio with only a modest increase of the scheduling overhead. FF in contrast to NF considers all partially filled bins as possible destinations for the item to be packed. An item is placed into the first (lowest indexed) bin into which it will fit. If no such bin exists, a previously empty bin is opened and the item is placed into this bin. It has been shown [21] that FF has time-complexity $\Theta(n \log n)$ for a list of n items.

Theorem 4. Level(FF) *is* 2.7-*competitive.*

The proof of this theorem uses the weighting function from [16]. Let $W : [0,1] \to [0, 8/5]$ be defined as follows (cf. Fig. 3):

$$W(\alpha) = \begin{cases} \frac{6}{5}\alpha & \text{for } 0 \leqslant \alpha \leqslant \frac{1}{6}, \\ \frac{9}{5}\alpha - \frac{1}{10} & \text{for } \frac{1}{6} < \alpha \leqslant \frac{1}{3}, \\ \frac{6}{5}\alpha + \frac{1}{10} & \text{for } \frac{1}{3} < \alpha \leqslant \frac{1}{2}, \\ \frac{6}{5}\alpha + \frac{4}{10} & \text{for } \frac{1}{2} < \alpha \leqslant 1. \end{cases}$$

We also need the following results from [16]:

Lemma 5. *Let B denote a set of items with total size $\leqslant 1$. Then*

$$\sum_{b \in B} W(\text{size}(b)) \leqslant \frac{17}{10}.$$

If all sizes are $\leqslant 1/2$,

$$\sum_{b \in B} W(\text{size}(b)) \leqslant \frac{3}{2}.$$

Theorem 6. *If L is a list of items with sizes $\leqslant 1$,*

$$\text{FF}(L) < \sum_{x \in L} W(\text{size}(x)) + 1.$$

Together with the above lemma this theorem provides the best-known upper bound for the number of bins used by first-fit. If L^* is the number of bins used in an optimal packing of L, first-fit uses at most $\lceil (17/10)L^* \rceil$ bins. Now we are ready to prove Theorem 4:

Proof. Let $\mathscr{J}$ be a job system with unit execution time and arbitrary precedence-constraints. We define

$$\bar{W}(\mathscr{J}) = \sum_{j \in \mathscr{J}} W(\text{size}(j)).$$

Thus $\bar{W}(\mathscr{J})$ is the total weight of all job sizes. Let l be the number of levels of the job system. For $1 \leqslant i \leqslant l$ let U_i be the set of jobs of each level. By Theorem 6 we can upper bound the length of the partial schedule for each level i, $1 \leqslant i \leqslant l$, generated by Level(FF):

$$T_{\text{LEVEL(FF)}}(U_i) < \bar{W}(U_i) + 1.$$

We can think of an optimal packing of $\mathscr{J}$ with the dependencies removed as a partition of $\mathscr{J}$ into $\mathscr{J}^*$ sets each of which has total size $\leqslant 1$. Applying Lemma 5 yields $\bar{W}(\mathscr{J}) \leqslant \frac{17}{10}\mathscr{J}^*$. Together with the fact that the length of the optimal schedule for $\mathscr{J}$ without dependencies cannot be longer than the length of the optimal schedule for $\mathscr{J}$ we conclude:

$$T_{\text{LEVEL(FF)}} = \sum_{i=1}^{l} T_{\text{LEVEL(FF)}}(U_i) < \sum_{i=1}^{l} (\bar{W}(U_i) + 1) = \bar{W}(\mathscr{J}) + l \leqslant 1.7\, T_{\text{opt}} + l.$$

Since $l = T_{\max} \leqslant T_{\text{opt}}$, the result follows. □

The competitive ratio 2.7 of Level(FF) is nearly optimal. To show this, we give an asymptotic lower bound of 2.691 for the competitive ratio of each deterministic on-line

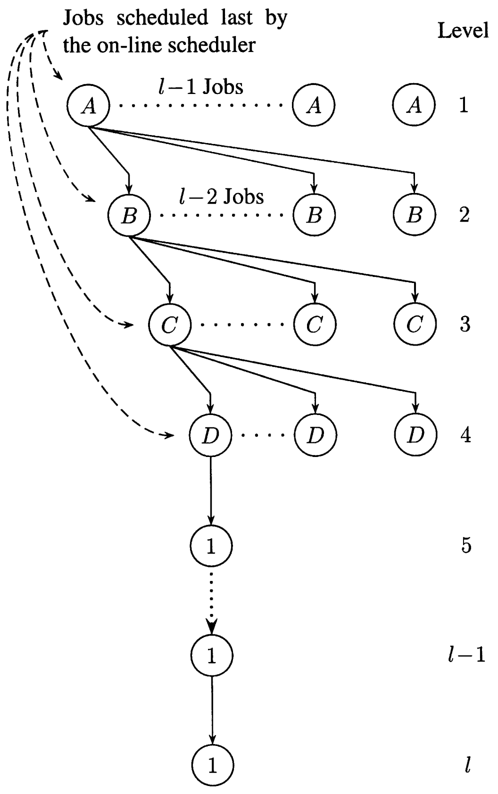

Fig. 4. Job system used in lower bound proof.

scheduling algorithm. For the sake of clarity, we first prove a slightly weaker lower bound of $2.69T_{\mathrm{opt}} - 4$ for the length of a schedule generated by a deterministic on-line scheduler. Using Salzer numbers we refine this construction to derive the asymptotic lower bound.

Fix $N \in \mathbb{N}$, $N \geqslant 7 \cdot 1806$, and let

$$A := \left\lfloor \frac{N}{2} \right\rfloor + 1, \qquad B := \left\lfloor \frac{N}{3} \right\rfloor + 1,$$

$$C := \left\lfloor \frac{N}{7} \right\rfloor + 1, \qquad D := N - A - B - C - 1.$$

The job system (see Fig. 4) consists of $l \geqslant 4$ levels with one chain of $l-4$ tasks (i.e., jobs that require one processor for execution) and l jobs of size A, $l-1$ jobs of size B, $l-2$ jobs of size C, $l-3$ jobs of size D.

Additional dependencies are assigned dynamically by an adversary depending on which parallel job of each level is scheduled last by the on-line algorithm. This is possible because the on-line scheduler cannot distinguish between the parallel jobs on the same level. The optimal schedule has length l and is shown in Fig. 5. Here, the

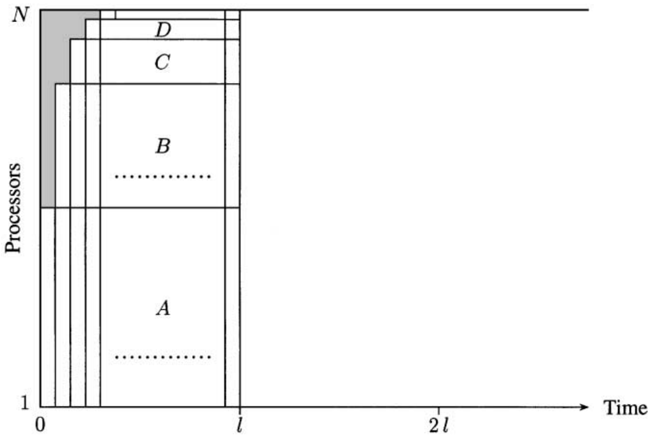

Fig. 5. Optimal schedule.

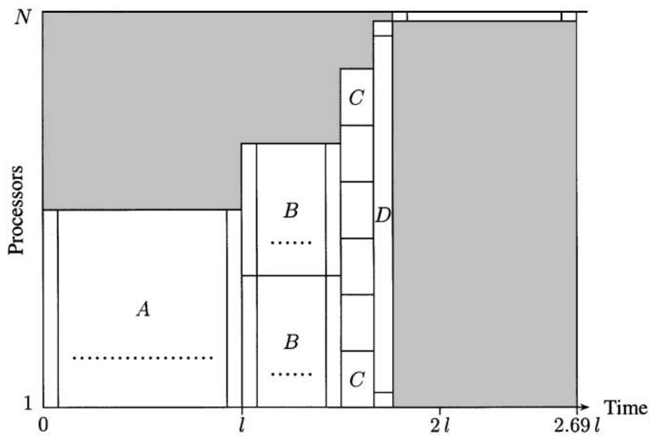

Fig. 6. On-line schedule generated by Level(FF).

parallel job with successors is scheduled first on each level. Contrary to the optimal solution, the on-line scheduler is forced to schedule and execute all jobs on one level to make the jobs on the next level available. The schedule generated by Level(FF) is thus the best possible on-line schedule (see Fig. 6) and has length

$$l + \left\lceil \frac{l-1}{2} \right\rceil + \left\lceil \frac{l-2}{6} \right\rceil + \left\lceil \frac{l-3}{42} \right\rceil + (l-4) > 2.69\, l - 4,$$

if $2 \nmid (l-1)$, $6 \nmid (l-2)$, and $42 \nmid (l-3)$. It is easy to see that any $l \in \mathbb{N}$ with $42 \mid l$ fulfills the above conditions.

The following sequence $(t_i)_{i \in \mathbb{N}}$ was investigated by Salzer [24]:

$$t_1 = 2,$$
$$t_{i+1} = t_i(t_i - 1) + 1 \quad \text{for } i \geqslant 1.$$

The first five numbers of this sequence are 2, 3, 7, 43, 1807. Closely related is the following series:

$$h_\infty = \sum_{i=1}^{\infty} \frac{1}{t_i - 1} > 1.69103. \tag{1}$$

There are two basic relations for the Salzer numbers that can be derived inductively from their definition:

$$\sum_{i=1}^{k} \frac{1}{t_i} + \frac{1}{t_{k+1} - 1} = 1, \quad \prod_{i=1}^{k} t_i = t_{k+1} - 1.$$

Let $A_i = \lfloor N/t_i \rfloor + 1$, $1 \leqslant i \leqslant k$, be the sizes of the parallel jobs on the first k levels. Setting $A_{k+1} = N - \sum_{i=1}^{k} A_i - 1$, we can conclude that

$$A_{k+1} < \frac{N}{t_{k+1} - 1} - 1, \quad A_{k+1} \geqslant \frac{N}{t_{k+1} - 1} - (k+1).$$

It is easy to see that $t_{k+1} - 1$ jobs of size A_{k+1} can be scheduled in one time-step on $N - 1$ processors. To ensure that no more than $t_{k+1} - 1$ jobs of size A_{k+1} can be co-scheduled on N processors we choose $N > (k+1)(t_{k+2} - 1)$. The job system again consists of $l \geqslant k+1$ levels with one chain of $l - (k+1)$ tasks and $l - (i-1)$ jobs of size A_i, $1 \leqslant i \leqslant k+1$. Dependencies are assigned dynamically as above. The length of the optimal schedule is l, whereas every schedule generated by a deterministic on-line scheduler has length at least

$$\sum_{i=1}^{k+1} \left\lceil \frac{l - (i-1)}{t_i - 1} \right\rceil + l - (k+1).$$

From this and (1) we see that the competitive ratio can be brought arbitrarily close to $1 + h_\infty$ for $k \to \infty$, $l = \omega(k)$.

The competitive ratio of Level(FF) can be improved if the maximum size of a job is restricted to $\lfloor N/2 \rfloor$:

Theorem 7. Level(FF) *is* 2.5-*competitive, if no job requests more than half of the total number of processors.*

Proof. Analogous to the proof of Theorem 4 using the second inequality of Lemma 5. □

The same bound holds if the next-fit-decreasing (NFD) bin-packing heuristic (presort the items in non-increasing order, then use NF) is used instead of FF. This follows easily form the fact that the average efficiency of 2 consecutive time-steps in a partial schedule for a level generated by NFD is $> 2/3$ in this case.

Similarly to the unrestricted case, an asymptotic lower bound > 2.4 for the competitive ratio of any deterministic on-line scheduler for this problem can be derived.

Further restrictions of the maximum job size might yield somewhat better competitive ratios for the LEVEL algorithm, but this situation is already handled well by the GENERIC algorithm in [13, 25] which achieves competitive ratio $1 + 1/(1 - \lambda)$, if no job requests more than λN, $0 < \lambda < 1$, processors. For example, $\lambda = 1/2$ yields competitive ratio 3 for the GENERIC algorithm that is valid for job systems with arbitrary execution times.

We also remark that the results of this subsection remain valid if we assume a one-dimensional array of length N as interconnection topology instead of using the complete model, since the BIN PACKING algorithms assign consecutive processors to the jobs and the assignments in different time-steps are independent from each other.

4.2. Hypercube

In this subsection we study the problem of on-line scheduling parallel job systems with arbitrary precedence-constraints and unit execution times for hypercube connected parallel computers.

It is not difficult to schedule a set of independent parallel jobs each of which requests a subcube of a certain dimension. First, we sort the jobs by size in non-increasing order. To avoid fragmentation, we use only normal subcubes for job execution:

Definition 8. A k-dimensional subcube is called *normal*, if the labels of all its processors differ only in the last k positions.

For each time-step of our schedule we allocate jobs from the head of the sorted list to normal subcubes while there are unscheduled jobs left and the hypercube is not completely filled. If the time-step is full, we have to add a new time-step to our schedule (if there are any unscheduled jobs left).

It is easy to see that the efficiency of this schedule for independent jobs is 1 in all time-steps except possibly the last. We refer to this strategy as PACK_HC. The algorithm for job systems with arbitrary dependencies is just the LEVEL algorithm using PACK_HC instead of a BIN PACKING heuristic (cf. Fig. 7).

Theorem 9. LEVEL_HC *is an optimal deterministic on-line scheduler with competitive ratio* 2.

Proof. The number of iterations of the while-loop is exactly the length of a critical path in the dependency graph. Thus $T_{<1} \leqslant T_{\max} \leqslant T_{\text{opt}}$. Since the efficiency of the schedule is at least $1/N$ all the time, we have for fixed N by Lemma 1:

$$T_{\text{LEVEL\_HC}} \leqslant \left(1 + \frac{1 - 1/N}{1}\right) T_{\text{opt}} = \left(2 - \frac{1}{N}\right) T_{\text{opt}}.$$

It remains to show that no deterministic on-line scheduler can achieve a better competitive ratio. To this behalf, we use a job system similar to the preceding subsection.

```
Algorithm LEVEL_HC:
begin
    while not all jobs are finished do
        begin
        $\mathscr{A} := \{J \in \mathscr{J} \mid J$ is available$\}$;   // next EPT level
        schedule all jobs in $\mathscr{A}$ according to PACK_HC;
        wait until all scheduled jobs are finished;
        end;
end.
```

Fig. 7. The LEVEL_HC algorithm.

It uses $N-1$ levels with $N+1$ tasks on each level. Again, the dependencies are assigned dynamically by an adversary according to the decisions of the deterministic on-line scheduler. The task from level i, $1 \leqslant i \leqslant N-2$, scheduled last by the on-line scheduler is designated to be predecessor of all tasks on level $i+1$. Therefore, any on-line scheduler ALG needs at least 2 time-steps to schedule all tasks of one level. In an optimal schedule, the task with dependencies is scheduled first together with $N-1$ other tasks from the same level. The $N-1$ remaining tasks are scheduled in time-step N. This gives the desired lower bound for the competitive ratio:

$$\frac{T_{\mathrm{ALG}}}{T_{\mathrm{opt}}} \geqslant \frac{2(N-1)}{N} = 2 - \frac{1}{N}. \qquad \square$$

The job system in the proof of Theorem 9 contains no parallel jobs and the hypercube structure is not used at all. Therefore, the derived lower bound is valid for any interconnection topology and sequential job systems as well as parallel job systems. Note that the structure of the dependency graph is an out-forest.

Corollary 10. *No deterministic on-line algorithm for scheduling job systems with unit execution times and dependencies can have a competitive ratio better than* 2.

This result has been obtained independently by Epstein [9]. Interestingly, the above lower bound is identical to the lower bound proved by Shmoys et al. [26] for sequential job systems with arbitrary running times but without precedence-constraints.

5. Parallel job systems with restricted runtime ratio

We have shown in the preceding section that on-line scheduling of parallel jobs with unit execution time and precedence-constraints is possible with small constant competitive ratio. On the other hand, if execution times are arbitrary, there exists no on-line scheduler with acceptable worst-case performance. It is only natural to explore

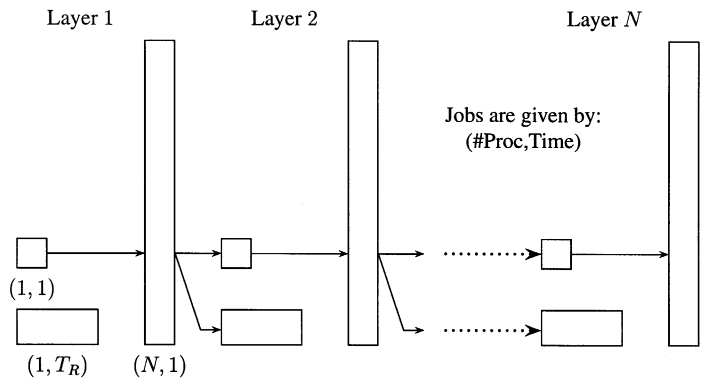

Fig. 8. Difficult job system for RRR-scheduling.

the case that job runtimes are restricted by some criterion other than unit execution time in order to achieve a respectable competitive ratio.

For a set of jobs $\mathscr{J}$ we therefore define the *runtime ratio* $\mathrm{RR}(\mathscr{J}) := t_{\max}/t_{\min}$. In this section we study the problem of on-line scheduling parallel job systems with dependencies where the runtime ratio is bounded from above by a parameter $T_\mathrm{R} \geqslant 1$ which is not known to the on-line scheduler. This problem often arises in practice when upper and lower bounds for the running time of a job are known in advance but the actual running time is unknown. This situation also makes clear that the parameter T_R cannot be used as additional information for scheduling decisions by the on-line scheduler and is therefore not part of the problem instance. Indeed, our results show that this knowledge is not necessary for the on-line scheduler to achieve a near optimal competitive ratio that depends only on T_R.

In this paper, we study this problem for the complete model. First, we give a lower bound of $\max\{(T_\mathrm{R}+1)/2, h_\infty + 1\}$ for the asymptotic competitive ratio of any deterministic on-line scheduler for this problem. For simplicity, we normalize the running time of the shortest job to 1. The job system used in this lower bound argument is very simple (see Fig. 8) and consists of N layers with two tasks and one parallel job of size N on each layer. The parallel job depends on one of the tasks on the same layer and is predecessor of both tasks of the following layer. The task scheduled first by the on-line scheduler is assigned running time T_R and the remaining task runs for 1 unit of time and is predecessor of the parallel job. Clearly, the makespan of any schedule generated by an on-line scheduler is at least $N(T_\mathrm{R}+1)$. If T_R is sufficiently large (e.g., $T_\mathrm{R} \geqslant 2$), the optimal solution first schedules the critical path which has length $2N$ followed by the tasks of length T_R in parallel. The competitive ratio of any deterministic on-line scheduler is thus lower bounded by

$$\frac{N(T_\mathrm{R}+1)}{2N+T_\mathrm{R}} \underset{N\to\infty}{\longrightarrow} \frac{T_\mathrm{R}+1}{2}.$$

```
**Algorithm** RRR
**begin**
    **while** $L_1 \neq \emptyset$ **do**
        schedule a big job exclusively;
    **while** not all jobs are finished **do**
        **begin**
        **while** $L_2 \neq \emptyset$ **do**
            schedule small jobs greedily;
        **if** $L_1 \neq \emptyset$ **then**
            **if** a big job can be scheduled **then**
                do it;
            **else**
                **if** $\alpha \geqslant 1/2$ **then**
                    wait for a scheduled job to finish;
                **else**  // *start of a delay phase*
                    collect small jobs that become available
                    during the next 2 units of time;
                    schedule those jobs greedily and
                    then wait for all scheduled jobs to finish;
                    **while** $L_1 \neq \emptyset$ **do**
                        schedule a big job exclusively;
                **fi**;
            **fi**;
        **else**
            wait for next available job;
        **fi**;
        **end**;
**end.**
```

Fig. 9. The RRR algorithm.

For small T_R, this bound is quite weak. But in this case, we can use the job system from the lower bound construction in Section 4.1. Since the runtime ratio of this job system is 1 this yields an asymptotic lower bound of $h_\infty + 1$ for the competitive ratio.

We now describe an algorithm designated RRR (see Fig. 9) that achieves competitive ratio $T_\mathrm{R}/2 + 4$. A key feature of this algorithm is the distinction between *big* jobs that request more than half of the total number of processors and *small* jobs with size $\leqslant \lfloor N/2 \rfloor$. Let $\alpha := \alpha(t)$ denote the efficiency at time t. The RRR algorithm tries to keep the efficiency at least 1/2 whenever possible. There are two reasons that hinder the RRR algorithm from achieving this goal. First, there might be no job available and second, there might be not enough processors available to schedule a big job.

The second case is much more severe than the first one which can be handled by the GRAHAM argument (cf. Lemma 2) without much loss of performance. Therefore, the RRR algorithm must prevent big jobs from being delayed too long in order to bound the fraction of the total schedule length with low efficiency. This is done by occasionally stopping to schedule small jobs, if all big jobs request more processors than currently available and the efficiency is below 1/2.

We present two versions of the RRR algorithm. The first one assumes that $t_{\min}$ is a known quantity. Again, we normalize the running time of the shortest job to 1 and a *unit of time* refers to this normalized time quantum. In the second version we remove this assumption and employ an adaptive waiting-strategy to maintain a comparable competitive ratio. The RRR algorithm maintains two sets, L_1 and L_2, containing the available big and respectively small jobs. We assume that any job that becomes available is immediately inserted into the appropriate set, and we will not state this activity explicitly in the pseudo-code description of our algorithms.

Theorem 11. *The RRR algorithm is* $(T_{\mathrm{R}}/2+4)$*-competitive for any job system* $(\mathscr{J},\prec)$ *with* $\mathrm{RR}(\mathscr{J}) \leqslant T_{\mathrm{R}}$.

Proof. We partition the schedule generated by the RRR algorithm into 3 different kinds of phases:

1. Efficiency is at least 1/2.
2. Efficiency is below 1/2 and there is no job available.
3. Efficiency is below 1/2 and the algorithm waits for the termination of all jobs.

We refer to the third type as a *delay phase* and denote the total time of each kind by $T_{\geqslant 1/2}$, T_{nojob}, and T_{delay}, respectively. The total time of the RRR schedule that is spent in phases of types 1 and 2 can easily be bounded by $3T_{\mathrm{opt}}$, because we have $T_{\geqslant 1/2} \leqslant 2T_{\mathrm{opt}}$ by a straightforward area-argument and $T_{\mathrm{nojob}} \leqslant T_{\max} \leqslant T_{\mathrm{opt}}$ by Lemma 2.

It remains to show that $T_{\mathrm{delay}} \leqslant (T_{\mathrm{R}}/2 + 1)T_{\mathrm{opt}}$. We define a *delayed job* as a big job that was available at the beginning of a delay phase. Let t_i denote the start time of delay phase i. First, we bound the length of a delay phase by $T_{\mathrm{R}} + 2$. If no small jobs become available during the first two units of time after the beginning of a delay phase, no more jobs are scheduled until all currently running jobs terminate. Since the running time of any job is no more than T_{R}, such a delay phase lasts at most time T_{R}. On the other hand, if small jobs become available during the first two units of time, these are collected and scheduled greedily at time $t_i^s = t_i + 2$ (resp. $t_i^s < t_i + 2$ if all jobs running at time t_i terminate before two units of time have elapsed) in addition to those jobs still running at time t_i^s. If the total size of these small jobs (i.e., the total number of processors that all these small jobs request) is no more than the number of idle processors at time t_i^s, they can be scheduled immediately. Clearly, the length of a delay phase is bounded by $T_{\mathrm{R}} + 2$ in this case. Should the total size of the small jobs exceed the number of idle processors at time t_i^s we can schedule enough small jobs to raise the efficiency above 1/2 as long as small jobs that were collected during the interval $[t_i, t_i^s]$ are available. The time-span while the efficiency is at least 1/2 is, of course, a phase of type 1 and not part of the delay phase. Clearly, the length of the second part of a delay phase is bounded by T_R and therefore the length of a delay phase is always bounded by $T_R + 2$.

Let d denote the number of delay phases in a schedule generated by the RRR algorithm. We distinguish two cases:

1. $d = 1$: We have to show that the optimal solution needs at least time 2. This follows immediately from the fact that each delayed job must have a predecessor in the job system because otherwise it would have been scheduled earlier.
2. $d > 1$: This case will be proven by constructing a chain of jobs in the dependency graph with total execution time at least $2d$. From that we have $T_{opt} \geqslant 2d$ and together with $T_{delay} \leqslant d(T_R + 2)$ the claim follows.

The construction of this chain proceeds as follows: Starting with an arbitrary delayed job that is scheduled after delay phase d we observe that there must be a small job that is ancestor of this delayed job and is available immediately after the delayed jobs of delay phase $d-1$ (i.e., without having a small job as direct predecessor that is itself scheduled after the delayed jobs of delay phase $d - 1$) because otherwise this delayed job would have been scheduled earlier. We add such a small job at the front of the chain.

To augment the chain, we state the possibilities for the direct predecessor of a small job that is scheduled by the RRR algorithm immediately after the delayed jobs of delay phase i:

Type 1: Delayed job of delay phase i or
big job that is successor of a delayed job of delay phase i,
Type 2: Small job collected during delay phase i,
Type 3: Small job running from the beginning of delay phase i.

This is due to the fact that the RRR algorithm schedules all small jobs that are available by time t_i^s before the delayed jobs of delay phase i.

We continue the construction inductively according to these three possibilities. If there is a direct predecessor of Type 1 of the small job that is currently head of the list, we can repeat the initial construction step of the chain and add a delayed job and its small ancestor at the front of the chain. When there is no direct predecessor of Type 1 but a direct predecessor of Type 2, we add 2 more jobs at the front of the chain: the Type 2 job and a direct predecessor of this job that was running at the beginning of the delay phase during which this Type 2 job was collected. Finally, if there is only a direct predecessor of Type 3, we add this job at the front of the chain. The inductive construction stops as soon as the head of the chain is a small job that is scheduled before the delayed jobs of the first delay phase.

To complete the proof, we show that the total execution time of the jobs along this chain is at least $2d$. The construction of the chain starts with 2 jobs, a delayed job and its small ancestor. Since the minimum running time of any job is 1, these 2 jobs need at least 2 units of time for execution in any schedule. If the construction proceeds by adding a Type 1 job, the same argument applies. Continuing with a Type 2 job means that again 2 more jobs were added to the chain. If a Type 3 job is encountered, we know that this job must have execution time at least 2 because it is direct predecessor of a small job that is scheduled immediately after the delayed jobs of the delay phase the Type 3 job belongs to. Thus, for each delay phase in the schedule generated by the RRR algorithm, the above construction adds jobs with total execution time at least 2 to the chain. □

The assumption that $t_{\min}$ is known to the RRR algorithm can be dropped by employing an adaptive waiting strategy without much loss in competitive performance. We describe this adaptive version separately in order to keep our presentation modular. The modifications of the RRR algorithm are as follows (see also Fig. 10): Since $t_{\min}$ is now unknown the RRR_Adaptive algorithm does not collect small jobs during the first delay phase. In all following delay phases (if any), the algorithm calculates $t^i_{\min}$, the minimum execution time of any finished job up to the start of delay phase i. The duration during which small jobs are collected is now limited by $2\,t^i_{\min}$ (and, of course, by $t_{\max}$).

Theorem 12. *The* RRR_Adaptive *algorithm is* $(T_R/2 + 5.5)$*-competitive for any job system* $(\mathcal{J}, \prec)$ *with* $\mathrm{RR}(\mathcal{J}) \leqslant T_R$.

Proof. With the notation of the proof of Theorem 11 we conclude analogously that the above theorem holds for $d = 1$. If $d > 1$, we have

$$T_{\text{delay}} \leqslant d t_{\max} + 2\sum_{i=2}^{d} t^i_{\min}.$$

First, we show that $2\sum_{i=2}^{d} t^i_{\min} \leqslant 2T_{\text{opt}} - 2t_{\min}$. To see this, we observe that after delay phase i, $1 \leqslant i \leqslant d$, at least one delayed job has to be scheduled. Let $t^{d+1}_{\min} := t_{\min}$. The running time of such a delayed job is at least $t^{i+1}_{\min}$, since this job is executed before the start of delay phase $i+1$ (if $i<d$). Even in an optimal schedule all delayed jobs must be scheduled sequentially because they require more than half of the available processors for execution. Therefore,

$$2T_{\text{opt}} \geqslant 2\sum_{i=2}^{d+1} t^i_{\min} = 2\sum_{i=2}^{d} t^i_{\min} + 2t_{\min}. \tag{2}$$

As in the proof of Theorem 11 we can construct a chain of jobs in the dependency graph with total execution time at least $(2d-1)t_{\min}$. The only difference in the construction is that there is no collection of small jobs during the first delay phase and therefore a Type 3 job might only run for time $t_{\min}$ in this delay phase. This yields

```
Algorithm RRR_ADAPTIVE
begin
   $i := 0$;   // $i$ counts the number of delay phases
   while $L_1 \neq \emptyset$ do
      schedule a big job exclusively;
   while not all jobs are finished do
      begin
      while $L_2 \neq \emptyset$ do
         schedule small jobs greedily;
      if $L_1 \neq \emptyset$ then
         if a big job can be scheduled then
            do it;
         else
            if $\alpha \geqslant 1/2$ then
               wait for a scheduled job to finish;
            else   // start of a delay phase
               if $i > 0$ then
                  $i := i + 1$;
                  $t^i_{\min}$ := current minimum execution time;
                  collect small jobs that become available
                  for time $\leqslant 2\,t^i_{\min}$;
                  schedule those jobs greedily and
                  then wait for all scheduled jobs to finish;
               else
                  $i := i + 1$;
                  wait for all scheduled jobs to finish;
               fi;
               while $L_1 \neq \emptyset$ do
                  schedule a big job exclusively;
            fi;
         fi;
      else
         wait for next available job;
      fi;
      end;
end.
```

Fig. 10. The RRR_ADAPTIVE algorithm.

another lower bound for the optimum schedule length:

$$T_{\text{opt}} \geqslant (2d-1)t_{\min}. \tag{3}$$

From (2) and (3) we conclude

$$\begin{aligned} T_{\text{delay}} &\leqslant dt_{\max} + 2\sum_{i=2}^{d} t_{\min}^{i} \\ &\leqslant dt_{\max} + 2T_{\text{opt}} - 2t_{\min} \\ &\leqslant \frac{(d-1/2)T_{\text{R}}}{2d-1}T_{\text{opt}} + \frac{t_{\max}}{2} + 2T_{\text{opt}} \\ &\leqslant \left(\frac{T_{\text{R}}}{2} + \frac{5}{2}\right)T_{\text{opt}}. \end{aligned}$$

If the number of delay phases of a schedule generated by the RRR_ADAPTIVE algorithm is less than $(T_{\text{R}}+1)/2$, we can derive a better upper bound

$$T_{\text{delay}} \leqslant (d+2)T_{\text{opt}}.$$

However, this bound is useful for a posteriori analysis only, since the number of delay phases can be arbitrarily large. Since the total schedule time that is spent in phases of types 1 and 2 (cf. proof of Theorem 11) is bounded by $3T_{\text{opt}}$, the proof is complete. □

Clearly, both algorithms can easily compute the runtime ratio $\text{RR}(\mathscr{J})$ for any scheduled job system $\mathscr{J}$. From this, we can bound the actual performance for the generated schedules:

$$T_{\text{RRR}} \leqslant (\text{RR}(\mathscr{J})/2 + 4)T_{\text{opt}},$$

$$T_{\text{RRR\_ADAPTIVE}} \leqslant (\text{RR}(\mathscr{J})/2 + 5.5)T_{\text{opt}}.$$

For practical purposes it is desirable to have tools that allow to control the performance of a scheduler in addition to worst-case guarantees such as the competitive ratio. Let T_{big} be the sum of the execution times of all big jobs in $\mathscr{J}$, and let W_{total} denote the total work of all jobs. Then we have the following lower bound for the length of an optimal schedule:

$$T_{\text{opt}} \geqslant \max\{W_{\text{total}}/N, T_{\max}, T_{\text{big}}\}.$$

Again, our on-line algorithms can compute W_{total} and T_{big} during the scheduling process. Assuming that the on-line scheduler has knowledge of the predecessor/successor relationships (which usually will be the case after all jobs have been scheduled), $T_{\max}$ can be computed by searching a longest path in the dependency graph. The quotient of the length of the on-line schedule and the above lower bound is then an upper bound for the performance of our on-line schedulers.

6. Conclusion and open problems

We have presented and analyzed several on-line scheduling algorithms for parallel job systems. It has become evident that *runtime restrictions* improve the competitive performance achievable by on-line schedulers. Therefore, if enough a priori knowledge on job running times is available to bound the runtime ratio of a job system, our schedulers can guarantee a reasonable utilization of the parallel system. But even without any such knowledge the RRR_Adaptive algorithm produces schedules that are almost best possible from a worst-case point of view. All on-line algorithms considered in this paper are computationally simple, and thus the scheduling over-head involved can safely be neglected, provided that the system has suitable means to deliver the necessary load information.

It still remains to study the described scheduling problems for a number of other popular interconnection topologies. In the unit execution time model, we have a 46/7-competitive algorithm for two-dimensional array [2], but in general it appears that the competitive ratio might grow exponentially with the dimension of the array.

Acknowledgements

We would like to thank Klaus Jansen for helpful suggestions for the analysis of the Level(FF) algorithm, and Jiří Sgall and an anonymous referee for their valuable comments.

References

[1] G.S. Almasi, A. Gottlieb, Highly Parallel Computing, 2nd revised ed., Benjamin/Cummings, Redwood City, CA, 1994.

[2] S. Bischof, Efficient algorithms for on-line scheduling and load distribution in parallel systems, Ph.D. Thesis, Institut für Informatik, Technische Universität München, 1999. Available on-line: http://wwwmayr.in.tum.de/berichte/1999/bischof-thesis.ps.gz.

[3] J. Błażewicz, M. Drabowski, J. Węglarz, Scheduling multiprocessor tasks to minimize schedule length, IEEE Trans. Comput. C-35 (5) (1986) 389–393.

[4] J. Błażewicz, K.H. Ecker, E. Pesch, G. Schmidt, J. Węglarz, Scheduling Computer and Manufacturing Processes, Springer, Berlin, 1996.

[5] A. Borodin, R. El-Yaniv, Online Computation and Competitive Analysis, Cambridge University Press, Cambridge, 1998.

[6] S. Chakrabarti, S. Muthukrishnan, Resource scheduling for parallel database and scientific applications, Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'96, ACM Press, New York, 1996, pp. 329–335.

[7] E.G. Coffman Jr., M.R. Garey, D.S. Johnson, Approximation algorithms for bin packing: a survey, in: D.S. Hochbaum (Ed.), Approximation Algorithms for NP-Hard Problems, PWS Publishing Company, Boston, 1996, pp. 46–93 (Chapter 2).

[8] J. Du, J.Y.-T. Leung, Complexity of scheduling parallel task systems, SIAM J. Discrete Math. 2 (1989) 473–487.

[9] L. Epstein, Lower bounds for on-line scheduling with precedence constraints on identical machines, in: K. Jansen, J. Rolim (Eds.), Proceedings of the First International Workshop on Approximation

Algorithms for Combinatorial Optimization APPROX'98, Lecture Notes in Computer Science, Vol. 1444, Springer, Berlin, 1998, pp. 89–98.

[10] D.G. Feitelson, L. Rudolph, Parallel job scheduling: issues and approaches, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing (IPPS'95 Workshop), Lecture Notes in Computer Science, Vol. 949, Springer, Berlin, 1995, pp. 1–18.

[11] D.G. Feitelson, L. Rudolph, Toward convergence in job schedulers for parallel supercomputers, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing (IPPS'96 Workshop), Lecture Notes in Computer Science, Vol. 1162, Springer, Berlin, 1997, pp. 1–26.

[12] D.G. Feitelson, L. Rudolph, U. Schwiegelshohn, K.C. Sevcik, P. Wong, Theory and practice in parallel job scheduling, in: D.G. Feitelson, L. Rudolph (Eds.), Job Scheduling Strategies for Parallel Processing (IPPS'97 Workshop), Lecture Notes in Computer Science, Vol. 1291, Springer, Berlin, 1997, pp. 1–34.

[13] A. Feldmann, M.-Y. Kao, J. Sgall, S.-H. Teng, Optimal on-line scheduling of parallel jobs with dependencies, J. Combin. Optim. 1 (4) (1998) 393–411.

[14] A. Feldmann, J. Sgall, S.-H. Teng, Dynamic scheduling on parallel machines, Theoret. Comput. Sci. (Special Issue on Dynamic and On-line Algorithms) 130 (1) (1994) 49–72.

[15] A. Fiat, G.J. Woeginger (Eds.), in: Online Algorithms: The State of the Art, Lecture Notes in Computer Science, Vol. 1442, Springer, Berlin, 1998.

[16] M.R. Garey, R.L. Graham, D.S. Johnson, A.C.-C. Yao, Resource constrained scheduling as generalized bin packing, J. Combin. Theory Ser. A 21 (1976) 257–298.

[17] M.R. Garey, D.S. Johnson, R.E. Tarjan, M. Yannakakis, Scheduling opposing forests, SIAM J. Algebraic Discrete Methods 4 (1) (1983) 72–93.

[18] M.N. Garofalakis, Y.E. Ioannidis, Parallel query scheduling and optimization with time- and space-shared resources, in: M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, M.A. Jeusfeld (Eds.), Proceedings of the 23rd International Conference on Very Large Data Bases VLDB'97, Morgan Kaufmann Publishers, San Francisco, CA, 1997, pp. 296–305.

[19] R.L. Graham, Bounds for certain multiprocessing anomalies, Bell System Tech. J. (1966) 1563–1581.

[20] R.L. Graham, Bounds on multiprocessing timing anomalies, SIAM J. Appl. Math. 17 (2) (1969) 416–429.

[21] D.S. Johnson, Fast algorithms for bin packing, J. Comput. System. Sci. 8 (1974) 272–314.

[22] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[23] E. Rahm, Dynamic load balancing in parallel database systems, in: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Proceedings of the Second International EURO-PAR Conference on Parallel Processing EURO-PAR'96, vol. I, Lecture Notes in Computer Science, Vol. 1123, Springer, Berlin, 1996, pp. 37–52.

[24] H.E. Salzer, The approximation of numbers as sums of reciprocals, Amer. Math. Monthly 54 (1947) 135–142.

[25] J. Sgall, On-line scheduling on parallel machines, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.

[26] D.B. Shmoys, J. Wein, D.P. Williamson, Scheduling parallel machines on-line, SIAM J. Comput. 24 (6) (1995) 1313–1331.

[27] D.D. Sleator, R.E. Tarjan, Amortized efficiency of list update and paging rules, Comm. ACM 28 (2) (1985) 202–208.

[28] C.B. Stunkel, D.G. Shea, B. Abali, M. Atkins, C.A. Bender, D.G. Grice, P.H. Hochschild, D.J. Joseph, B.J. Nathanson, R.A. Swetz, R.F. Stucke, M. Tsao, P.R. Varker, The SP2 communication subsystem, Research Report RC 19914, IBM Research Division, T.J. Watson Research, 1994.

[29] Q. Wang, K.H. Cheng, A heuristic of scheduling parallel tasks and its analysis, SIAM J. Comput. 21 (2) (1992) 281–294.