On-Line Scheduling of Parallel Jobs with Runtime Restrictions

Stefan Bischof and Ernst W. Mayr

Institut für Informatik
Technische Universität München
D-80290 München, Germany

{bischof|mayr}@in.tum.de
http://wwwmayr.in.tum.de/

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 in this paper have optimal competitive ratio up to small additive constants.

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 [2]. But in many situations the problem arises to find a schedule for a set of parallel jobs [4, 5, 2].

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 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 [6, 9] 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 (cf. [1]), our algorithm is almost optimal.

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). 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 $T_R \geq 2$ 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 RRR (Restricted Runtime Ratio) 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 $\leq T_R$. 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. In fact, the modified algorithm for this third model is $T_R/2 + 5.5$ competitive (see [1] for details).

The remainder of this paper is organized as follows. In Section 2 we introduce our scheduling model, some notation and definitions. 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 $\mathcal{J} = \{J_1, J_2, \ldots, J_m\}$ 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 $\mathcal{J}$ given as a partial order $\prec$ on $\mathcal{J}$. If $J_a \prec J_b$ then J_b cannot be scheduled for execution before J_a is completed. A *task* is a job that requires one processor for execution.

A *schedule* for a job system $(\mathcal{J}, \prec)$ 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. Therefore, the various types of interconnection networks have to be treated separately. In this paper we study the *complete model* which assumes that a job J_a requests n_a processors ($1 \leq n_a \leq N$) for execution and any subset of processors of size n_a may be allocated. Results for hypercubes can be found in [1].

It is always possible to transform a job system $(\mathcal{J}, \prec)$ into a directed acyclic graph $D = (\mathcal{J}, E)$ 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 $(\mathcal{J}, \prec)$. 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 $(\mathcal{J}, \prec)$ 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 $(\mathcal{J}, \prec)$. 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 *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. [9, 6]) for a given job system $(\mathcal{J}, \prec)$:

Table 1. Frequently used notations

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

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. A deterministic on-line algorithm ALG is called *c-competitive* if $T_{\text{ALG}} \leq cT_{\text{opt}}$ 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.

3 Previous and Related Work

Extensive work on non-preemptive on-line scheduling of parallel jobs with or without dependencies was done by FELDMANN, KAO, SGALL and TENG [6, 9, 7]. 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 $1 + \frac{1}{1-\lambda}$. Setting $\lambda = 1/2$, for example, yields a 3-competitive algorithm. Another possible 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 $2+\Phi$, where $\Phi = (\sqrt{5}-1)/2$ is the golden ratio. The two approaches just described can be combined to yield an optimal on-line scheduler with competitive ratio $2+\frac{\sqrt{4\lambda^2+1}-1}{2\lambda}$ 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 [6, 9] 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. It turns out that the problem becomes manageable with small constant competitive ratio even if nothing is known about the precedence constraints.

To fill the gap between these two extremes — totally unrelated running times versus unit execution times — we identify the runtime ratio as the distinctive parameter of a job system for the achievable competitive ratio. Our results for the proposed on-line schedulers in Section 5 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.

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, GRAHAM, JOHNSON and YAO [8]. We show that similar results hold in an on-line environment, where a job is available only if all its predecessors have completed execution.

The LEVEL algorithm (see Fig. 1) collects all jobs that are available from the beginning. Since available jobs are independent we can easily transform the problem 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 [3] 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.

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

Fig. 1. The LEVEL(PACK) algorithm

We now show that the First-Fit (FF) BIN-PACKING heuristic is a good choice for PACK. FF 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.

Theorem 1. LEVEL(FF) *is* 2.7*-competitive.*

The proof of this theorem uses the following results from [8] involving a piecewise linear weighting function $W : [0, 1] \to [0, 8/5]$.

Lemma 2. *Let B denote a set of items with total size ≤ 1. Then*

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

Theorem 3. *If L is a list of items with sizes ≤ 1, then*

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

where $\mathrm{FF}(L)$ denotes the number of bins used by First-Fit to pack L. Now we are ready to prove Theorem 1:

Proof. Let $\mathcal{J}$ be a job system with unit execution time and arbitrary dependencies. We define

$$\overline{W}(\mathcal{J}) = \sum_{j \in \mathcal{J}} W(\mathrm{size}(j)).$$

Thus $\overline{W}(\mathcal{J})$ is the total weight of all job sizes. Let l be the number of levels of the job system. For $1 \leq i \leq l$ let U_i be the set of jobs of each level. By Theorem 3 we can upper bound the length of the partial schedule for each level i, $1 \leq i \leq l$, generated by LEVEL(FF):

$$T_{\mathrm{LEVEL(FF)}}(U_i) < \overline{W}(U_i) + 1.$$

We can think of an optimal packing of $\mathcal{J}$ with the dependencies removed as a partition of $\mathcal{J}$ into $\mathcal{J}^*$ sets each of which has total size ≤ 1. Applying Lemma 2 yields $\overline{W}(\mathcal{J}) \leq \frac{17}{10}\mathcal{J}^*$. Together with the fact that the length of the optimal schedule for $\mathcal{J}$ without dependencies cannot be longer than the length of the optimal schedule for $\mathcal{J}$ we conclude:

$$T_{\mathrm{LEVEL(FF)}} = \sum_{i=1}^{l} T_{\mathrm{LEVEL(FF)}}(U_i) < \sum_{i=1}^{l} (\overline{W}(U_i) + 1) = \overline{W}(\mathcal{J}) + l \leq 1.7\, T_{\mathrm{opt}} + l.$$

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

Note that the results of this section remain valid if we assume a 1-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.

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 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 $\mathcal{J}$ we therefore define the *runtime ratio* $\mathrm{RR} = \mathrm{RR}(\mathcal{J}) := t_{\max}/t_{\min}$.

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 $T_R \geq 1$ 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.

First, we give a lower bound of $(T_R + 1)/2$ 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 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 $N(T_R + 1)$. If T_R is sufficiently large (e.g., $T_R \geq 2$), 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

$$\frac{N(T_R + 1)}{2N + T_R} \quad \xrightarrow[N \to \infty]{} \quad \frac{T_R + 1}{2}.$$

```
Algorithm RRR
begin
    while L_1 ≠ ∅ do schedule a big job exclusively;
    while not all jobs are finished do
    begin
        while L_2 ≠ ∅ do schedule small jobs greedily;
        if L_1 ≠ ∅ then
            if a big job can be scheduled then do it;
            elsif α ≥ 1/2 then
                wait for a scheduled job to finish;
            else   // start of a delay phase
            begin
                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 ≠ ∅ do schedule a big job exclusively;
            end;
        else wait for next available job;
    end;
end.
```

Fig. 2. The RRR algorithm

We now describe an algorithm designated RRR (see Fig. 2) that achieves competitive ratio $T_R/2 + 4$. 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 $\leq \lfloor N/2 \rfloor$. 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. 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 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 the RRR algorithm.

Theorem 4. *The* RRR *algorithm is* $(T_R/2+4)$*-competitive for any job system* $(\mathcal{J}, \prec)$ *and* $\mathrm{RR}(\mathcal{J}) \leq T_R$.

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 $T_{\geq 1/2}$, T_{nojob}, and T_{delay} respectively. The total time of the RRR schedule that is spent in phases of type 1 and 2 can easily be bounded by $3\,T_{\text{opt}}$, because we have $T_{\geq 1/2} \leq 2\,T_{\text{opt}}$ by a straightforward area-argument and $T_{\text{nojob}} \leq T_{\max} \leq T_{\text{opt}}$ by GRAHAM's critical path argument.

It remains to show that $T_{\text{delay}} \leq (T_R/2+1)T_{\text{opt}}$. 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_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, then 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 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_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 $T_{\mathrm{opt}} \geq 2d$ and together with $T_{\mathrm{delay}} \leq d(T_R + 2)$ 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 scheduled 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 with 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:

Theorem 5. *The* $\mathrm{RRR\_ADAPTIVE}$ *algorithm is* $(T_R/2 + 5.5)$*-competitive for any job system* $(\mathcal{J}, \prec)$ *and* $\mathrm{RR}(\mathcal{J}) \leq T_R$.

Proof. Omitted due to space restrictions (see [1]). □

6 Conclusion

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 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 overhead involved can safely be neglected, provided that the system has suitable means to deliver the necessary load information.

References

1. Stefan Bischof and Ernst W. Mayr. On-Line Scheduling of Parallel Jobs with Runtime Restrictions. SFB-Bericht 342/04/98 A, SFB 342, Institut für Informatik, Technische Universität München, April 1998. http://wwwmayr.in.tum.de/berichte/1998/TUM-I9810.ps.gz.
2. J. Błażewicz, K.H. Ecker, E. Pesch, G. Schmidt, and J. Węglarz. *Scheduling Computer and Manufacturing Processes*. Springer-Verlag, Berlin, 1996.
3. E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 2, pages 46–93. PWS Publishing Company, Boston, 1996.
4. Dror G. Feitelson and Larry Rudolph. Parallel Job Scheduling: Issues and Approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing (IPPS' 95 Workshop)*, LNCS 949, pages 1–18, Berlin, 1995. Springer-Verlag.
5. Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing (IPPS' 97 Workshop)*, LNCS 1291, pages 1–34, Berlin, 1997. Springer-Verlag.
6. Anja Feldmann, Ming-Yang Kao, Jiří Sgall, and Shang-Hua Teng. Optimal Online Scheduling of Parallel Jobs with Dependencies. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 642–651, New York, 1993.
7. Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science, Special Issue on Dynamic and On-line Algorithms*, 130(1):49–72, 1994.
8. M.R. Garey, R.L. Graham, D.S. Johnson, and A.C.-C. Yao. Resource Constrained Scheduling as Generalized Bin Packing. *J. Comb. Theory Series A*, 21:257–298, 1976.
9. Jiří Sgall. *On-Line Scheduling on Parallel Machines*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.