Clairvoyance versus cooperation in scheduling of independent tasks[⋆]

Tomas Plachetka

Comenius University, Bratislava, Slovakia
plachetka@fmph.uniba.sk

Abstract. *Finding a schedule with minimal makespan for a finite number of independent tasks on a homogeneous network of processors is an NP-hard problem if durations of all tasks are known. With only partial a-priori knowledge of tasks' time durations, it makes sense to look for online algorithms which guarantee short makespans in terms of small competitive ratios. Three algorithms are analysed and compared in this paper. The chunking algorithm assumes a-priori knowledge of the longest task's duration. The factoring algorithm assumes a-priori knowledge of the ratio between the longest and shortest task's durations. The work-stealing algorithm requires no a-priori knowledge but, unlike the previous two algorithms, requires a mechanism for redistribution of tasks which have already been assigned to processors. It turns out that work-stealing outperforms both the chunking and factoring algorithms when the number of tasks is sufficiently large. The analysis is not only asymptotic—it also provides an accurate (worst-case) prediction of makespans for all aforementioned algorithms for an arbitrary number of processors and tasks.*

1 Introduction

Finding an optimal schedule for a given number of independent tasks with known time durations on a given number of (equally fast) parallel processors is NP-hard [9]. An online version of this problem [14] assumes only a partial knowledge of tasks' durations. This online version appears in the same abstract form e.g. in parallel game-tree search, parallel ray tracing, scheduling of independent loops for multiprocessor machines etc. The challenge is to properly control the trade-off between the costs of work imbalance and communication.

The problem can be stated as follows. There is a *master* holding W pieces of indivisible work (*tasks*). Each task can be processed by any of N *workers* (the master can actually be one of the workers, playing both roles simultaneously). The durations of tasks can be different (e.g. processing of one task can take a second, processing of another task can take a minute). The workers are reliable, equally fast and are willing to complete the entire work as soon as possible (to minimise the *makespan*, i.e. the parallel time). The master's interest also is the fastest completion of all W tasks. The master and the workers are isolated from one another but can communicate via a reliable asynchronous postal service. The delivery of a packet (message) takes some time called *communication latency*. More precisely, latency is the time from the moment when a worker becomes idle until the moment when it receives some work to do (or realises that there is no more work to do). Latency is a constant which does not depend on the size of the packet (e.g. on number of tasks transferred in the packet). The master and the workers know this constant.[1]

If the master knows the durations of all tasks, it can compute the shortest schedule offline (solving an NP-hard problem) and send a single packet to each worker. The packet contains all the tasks assigned in the shortest schedule to that worker.

If the master knows nothing about the durations of all tasks, it can for example send one packet containing W/N arbitrarily chosen tasks to each worker. By doing so, the latency is added only once to the makespan (as in the previous offline algorithm). This algorithm is good in case of equal tasks' durations. But it can happen that all the tasks are very short except of W/N tasks which are very long. In a lucky case, each worker will receive a packet which contains one very long task and many short tasks. In an unlucky case, $N-1$ workers will receive packets which contain only short tasks, while one worker will receive a packet in which all the tasks are very long. Online analysis is interested in this worst case, where an "adversary" plays against the master and the workers. The intention of the adversary is to make the schedule as long as possible.

[⋆] This work was partially supported by the grant VEGA 1/0726/09.

[1] Note that in asynchronous message passing model the latency can vary not only for different runs of the same program with the same input, but even for different messages in the same run. It is bounded for a given run, but the bound is not a-priori known. In order to be fair by comparison of different algorithms, we assume that the latency is the same constant for all messages and all runs. This is a common assumption in publications relating to the problem in question. (In dedicated practical networks, the latency for a fixed message size is bounded—and a-priori known, as it can be measured in run-time before the actual computation begins.)

If the workers are not allowed to return tasks which they are assigned and if no information on tasks' durations is available, then no master's algorithm is intuitively "better" than the algorithm from the previous paragraph. For example, consider an algorithm where the master sends only one task as a reply to a request from an idle worker. Then the unlucky case is that all the packets are very short. The cost of communication can then be very high in comparison with the work itself and may exceed the total time of the previous algorithm.

All algorithms used in the previous examples use the same scheme. In the beginning, all the workers are idle. Each worker process runs a loop in which it sends a *job request* to the master process, waits for a *job* (a set of tasks sent in a message) and then processes all the tasks in the job, one after another. The algorithms only differ in how the master decides for the number of tasks (*job size*) which it sends when replying a job request. Without a knowledge on a specific task's duration only the job sizes K are important, not the choice of tasks in a job. A generic master's algorithm is shown in Fig. 1.

```
master_generic(int $W$, int $N$)
{
  int $K$;
  int $work = W$;
  while ($work > 0$)
  {
    wait for a job request from an idle worker;
    compute the job size $K$;
    assign a job consisting of $K$ yet unprocessed tasks
    to the idle worker;
    $work = work - K$;
  }
  reply job requests with $NO\_MORE\_WORK$;
}
```

Fig. 1. Generic assignment algorithm (without work redistribution).

Note that when a job request arrives, the master process must decide immediately how many yet unprocessed tasks it assigns in that job (as it might take long until another job request arrives). This decision is based on partial a-priori knowledge of tasks' durations. Only *deterministic online algorithms* will be considered, i.e. algorithms which do not internally use any source of randomness. Two of the three algorithms studied in this paper, chunking and factoring, follow the scheme from Fig. 1. The third algorithm, deterministic work stealing, uses a more complex scheme.

Chunking and factoring algorithms make use of partial information on the tasks' durations. They were first investigated in a probabilistic model, where tasks' durations are assumed to be realisations of a random variable with known mean and variance [11, 7, 6, 1, 10]. The goal in the probabilistic model is to find parameter settings which minimise the expected makespan of the algorithms. Optimal settings have not been found, only rough approximations are known.

In a deterministic model [12, 13] it is assumed that the information on maximal and minimal tasks' durations is available a-priori, i.e. before the computation begins. The goal in the cited papers was to find parameter settings which minimise the maximal makespan of chunking and factoring algorithms. The goal in this paper is to find parameter settings which minimise the competitive ratio of the algorithms.

The main contribution of this paper is competitive analysis and quantitative comparison of three algorithms in the deterministic model. Optimal parameters for chunking and factoring algorithms are derived. (As it turns out, the optimal parameter settings are almost the same in both scenarios—which is perhaps not surprising, as both scenarios focus on the worst-case input.) The third algorithm is deterministic work stealing. This algorithm requires no a-priori information and has no parameters; however, it requires a mechanism for redistribution of already assigned tasks. This means that the workers are allowed to communicate with one another and are allowed to return assigned, but yet unprocessed tasks to the master. We prove that the deterministic work stealing algorithm performs better than the previous two algorithms under certain assumptions which usually hold in practical systems. This makes deterministic work stealing very attractive for applications where no a-priori knowledge of tasks' durations is available.

The deterministic work stealing algorithm appears in the context of diffusive load balancing, e.g. in [3–5] (optimal load balancing scheme). In the context of diffusive load balancing the data locality is the main concern; the goal is to exploit the structure of the network in order to minimise the cost of a single work redistribution step. In this paper the network structure is ignored and the number of work redistribution steps is minimised.

The paper is organised as follows. Section 2 introduces the notation and definitions. Sections 3, 4 and 5 present online analysis of chunking, factoring and work stealing algorithms in the deterministic model (related results for chunking and factoring algorithms in a probabilistic model are briefly summarised in subsections). Performance of these algorithms is compared in Section 6. Section 7 concludes the paper.

2 Notation and definitions

The following notation is used throughout this paper:

N	number of worker processes; $N \geq 1$
W	total number of tasks (total work); $W \geq 1$
$\mathcal{M}$	makespan (total parallel time)
L	latency (duration of assignment of one job)
t_i	task's durations ($i = 1...W$); $t_i > 0.0$
T_{min}	minimal task's duration, $T_{min} = \min_{i=1\dots W} t_i$
T_{max}	maximal task's duration, $T_{max} = \max_{i=1\dots W} t_i$
T	ratio T_{max}/T_{min}; $T \geq 1.0$

In order to quantify the performance of algorithms, a standard definition of competitive ratio is used [2]:

Definition 1 (competitive ratio of algorithm). *For given W, N, L, let $\mathcal{CR}_{S(A)}$ denote the maximal ratio between the makespan $\mathcal{M}_S(A)$ of algorithm S which uses a-priori information A and the best offline makespan $\mathcal{M}_{\text{best\_offline}}$ over all sequences $t_1 \dots t_W$ of the input tasks' durations which conform to the a-priori information A:*

$$\mathcal{CR}_{S(A)}(W, N, L) = \sup_{t_1 \dots t_W} \frac{\mathcal{M}_{S(A)}(t_1 \dots t_W)}{\mathcal{M}_{\text{best\_offline}}(t_1 \dots t_W)}$$

where $\mathcal{M}_{S(A)}(t_1 \dots t_W)$ is the makespan of S with N processes on W tasks with durations $t_1 \dots t_W$, with assignment latency L. $\mathcal{CR}_{S(A)}$ is called competitive ratio of algorithm S with a-priori information A.

The best offline algorithm assigns all the tasks in one round, i.e. its communication overhead is L. It produces a schedule with the shortest makespan for Wtasks with durations $t_1 \dots t_W$. Thus competitive ratio of any algorithm is at least 1.0. The smaller it is, the better the algorithm is.

Definition 2 (comparison of algorithms). *An assignment algorithm S performs at least as well as an assignment algorithm R for some W, N, L (we will also say that S does not perform worse than R, or that S competes with R) iff*

$$\mathcal{CR}_{S(A)}(W, N, L) \leq \mathcal{CR}_{R(A')}(W, N, L)$$

In a practical setting the number of processes N is fixed (equal to the number of available processors). The latency L can be considered a constant. The number of tasks W is constant for a given run of an algorithm. Intuitively, the relative overhead of an efficient assignment algorithm should diminish with the growing number of tasks W, i.e. the competitive ratio should approach a small constant independent of N with increasing W.

Although we are particularly interested in asymptotic case $W \to \infty$, we prefer to keep the comparison parameterised with respect to W, N, L instead of using the limit values of $\mathcal{CR}_{S(A)}$ and $\mathcal{CR}_{R(A)}$ in Definition 2. This allows for a finer comparison of algorithms.

3 Chunking

The chunking algorithm [11], [8] always assigns jobs of size K to idling worker processes, where K remains constant (the last assignment may be an exception, where a smaller job is assigned), see Fig. 1. Once a job has been assigned to a worker, this decision cannot be changed—the worker must then compute all the tasks assigned in that job.

We will prove a general theorem which states that a-priori knowledge of T_{max} does not help much. The parallel time of any algorithm (including chunking) is in the worst case comparable with the sequential time for a sufficiently large number of tasks.

Theorem 1. *For all W, N, L, T_{max} such that $W > N^3 + N^2(N-1)T_{max}/L$ competitive ratio of an arbitrary assignment algorithm with no work redistribution and with a-priori knowledge of T_{max} is at least N (i.e. $\Omega(N)$ for $W = \Omega(N^3)$).*

Proof. Let $W > N^3 + N^2(N-1)T_{max}/L$. Let K denote the maximal job size assigned by an algorithm S. There are two cases:
case 1, $K \geq WL/(N^2L + N(N-1)T_{max})$;
case 2, $K < WL/(N^2L + N(N-1)T_{max})$.

In case 1, there is K^{int} such that $N \leq K^{int} \leq K$ and K^{int}/N is an integer. K^{int} tasks of the maximal job will have duration T_{max}, while all the other tasks will have duration $\varepsilon \to 0$. The best offline algorithm computes the K^{int} long tasks in parallel, whereas the algorithm S computes them sequentially. This implies (as S makes at least as many assignments as the best offline algorithm)

$$\mathcal{CR}_{S(T_{max})}(W, N, L) \geq \frac{K^{int} T_{max}}{K^{int} T_{max}/N} = N$$

In case 2, consider the latency overhead of the algorithm S, which is at least $WL/(NK)$. Assume that one task has duration T_{max} and is assigned in the last job; all the remaining tasks are of an equal duration $\varepsilon \to 0$). Hence the makespan of the algorithm S is at least $WL/(NK) + T_{max}$ and

$$\mathcal{CR}_{S(T_{max})}(W, N, L) \geq \frac{WL/(NK) + T_{max}}{L + T_{max}} \geq N$$

This completes the proof. □

Consider the case where all K tasks of some job are of duration $\varepsilon \to 0$ and all the other tasks are of duration T_{max}. The competitive ratio of the chunking algorithm using the job size K is then

$$\mathcal{CR}_{\text{chunking}(T_{max})}(W, N, L) = \frac{KT_{max} + WL/(NK)}{L + T_{max}}$$

The competitive ratio is minimised by setting the first derivative of $\mathcal{CR}_{\text{chunking}}$ with respect to K to zero and solving for K. This yields $K = \sqrt{\frac{WL(L+T_{max})}{NT_{max}}}$.

3.1 Chunking in a probabilistic model

Recall that the probabilistic model assumes that tasks' durations are realisations of a random variable with (known) mean μ and (known) standard deviation σ. The fixed-size chunking strategy in the probabilistic model was analysed by Kruskal and Weiss [11]. They derived the following estimation of the expected makespan $E[\mathcal{M}]$ for the chunk size K:

$$E[\mathcal{M}] \approx \frac{W}{N}\mu + \frac{WL}{NK} + \sigma\sqrt{2K\ln N} \qquad (1)$$

This formula has a nice intuitive interpretation. The first term is the time of executing W tasks on N processors on a system with no overhead. The second term describes the latency overhead. The third term describes the load imbalance due to the variation in tasks' durations. Unfortunately, the estimation in Equation 1 only holds if W and K are large and $K \gg \log N$. If these assumptions hold then also the optimal chunk size K^{opt} can be estimated:

$$\hat{K}^{opt} = \left(\frac{\sqrt{2}WL}{\sigma N\sqrt{\ln N}}\right)$$

If the assumptions above do not hold, [11] gives the following estimates for the expected makespan $E[\mathcal{M}]$:

$$E[\mathcal{M}] \approx \frac{W}{N}\mu + \frac{WL}{NK} + \sigma\sqrt{2K\ln\frac{\sigma N}{\sqrt{K}\mu}}$$

for $K \ll W/N$ and small $\sqrt{K}/N$; and

$$E[\mathcal{M}] \approx \frac{W}{N}\mu + \frac{WL}{NK} + \frac{N\sigma^2}{\mu}$$

for $K \ll W/N$ and large $\sqrt{K}/N$. However, a tractable analytical expression for the optimal chunk size K could not be derived.

4 Factoring

Factoring [7, 6, 1, 8] works in rounds, see Fig. 2 it could also be expressed in the form of the generic algorithm from Fig. 1 by rewriting the procedure `compute the job size` K, but doing so would make the algorithm more difficult for reading). In each round, it assigns N jobs of equal size. The job size is decreased after each round, whereby the job size in a round is a factor of the work remaining (the number of yet unassigned tasks) at the beginning of the round. The factor F remains constant over all rounds. During the last round, single-task jobs are assigned. Once a job has been assigned to a worker, the worker must compute all tasks assigned in that job.

```
MASTER_FACTORING(int $W$, int $N$, float $F$)
{
  int $K$;
  int $counter$;
  int $round = 0$;
  int $work = W$;
  while ($work > 0$)
  {
    $round = round + 1$;
    $K = \max(work/F, 1)$;
    $counter = 0$;
    while (($counter < N$) and ($work > 0$))
    {
      $counter = counter + 1$;
      wait for a job request from an idle worker;
      assign a job consisting of $K$ yet unprocessed tasks
      to the idle worker;
      $work = work - K$;
    }
  } reply job requests with $NO\_MORE\_WORK$;
}
```

Fig. 2. Factoring algorithm with factor F.

We will derive the optimal factor F, assuming that the ratio $T = T_{max}/T_{min}$ is the only a-priori knowledge available. (A similar analysis which assumes an a-priori knowledge of both T_{max} and T_{min} can be found in [8] and [12].) Denote K_i the job size which is assigned during the round i of the factoring algorithm and let w_i denote the number of still unassigned tasks at the beginning of round i. In order to be competitive, factoring guarantees that the longest sequential computation of a job of size K_i will not take longer than the shortest parallel computation of the still unassigned $w_i - K_i$ tasks on the remaining $N-1$ workers: $\text{max\_seq\_time}(K_i) \leq \text{min\_par\_time}(w_i - K_i, N-1)$. In order to minimise the assignment overhead, K_i must be as large as possible. The largest K_i which satisfies the inequality above (and thus guarantees the maximal imbalance of at most 1 task) is $K_i = w_i/(1 + T(N-1))$.

Note that it is only the assignment overhead which determines the competitive ratio of factoring. For example, the trick with setting durations of all the tasks of K_1 to T_{max} and computing them in parallel by the best offline algorithm does not work. The reason is that this does not increase the makespan of factoring at all: $K_1 T_{max} = WT_{max}/(T(N-1)) = WT_{min}/(N-1)$.

Theorem 2. *For all W, N, L, T competitive ratio of the factoring algorithm with a-priori knowledge of T using factor $F = 1 + T(N-1)$ is $O((\ln W)/W)$ and approaches 1 if $W \to \infty$.*

Proof. Let r denote the last round at the beginning of which the number of still unassigned tasks w_r is at most N (as the size of the jobs assigned in the round r is $K_r = 1$ and the number of the jobs assigned in the round r is at most N). It can be observed that the number of yet unassigned tasks w_i at the beginning of round i is equal to $w_i = W\,(1 - N/(1+T(N-1)))^{i-1}$. Solving $w_r \le N$ for maximal r yields the number of rounds r performed by the factoring algorithm:

$$r_{\text{factoring}} = \frac{\ln(W/N)}{\ln\frac{1+T(N-1)}{(N-1)(T-1)}} \tag{2}$$

The cost of assignments of the factoring algorithm is Lr. The imbalance of the factoring algorithm is at most 1 task. Thus the difference between the net makespan of the factoring (which does not include the cost of assignments) and the best net offline makespan is at most $T_{max} - T_{min}$. In the worst case all task's durations are T_{min} except of one task which is of duration T_{max} and is assigned in the last round of the factoring algorithm. Hence competitive ratio of the factoring algorithm can be bounded from above:

$$\begin{aligned}&\mathcal{CR}_{\text{factoring}(T)}(W,N,L) \le \\ &\le \frac{WT_{min}/N + T_{max} - T_{min} + L\left(\ln(W/N)/\ln\frac{1+T(N-1)}{(N-1)(T-1)}\right)}{WT_{min}/N + L} = \\ &= O((\ln W)/W)\end{aligned} \tag{3}$$

This completes the proof. □

4.1 Factoring in a probabilistic model

The factoring algorithm in a probabilistic model (with known μ and σ) was studied by Flynn, Flynn-Hummel and Schonberg in the context of scheduling independent loops on multiprocessor shared-memory machines. An approximation of the optimum job size $\hat{K}_i^{opt}$ which is used in round i was determined in [6, 7] by estimating the maximal portion of the remaining (unassigned) work which has a high probability of being completed by N processors within time $\mu w_i/N$. The analysis yields the following iteration scheme (at the beginning of round i, w_i denotes the number of still unassigned tasks, $1/(Nx_i)$ is the division factor):

$$w_1 = W,\ x_1 = 1 + \frac{N^2}{w_1}\left(\frac{\sigma}{\mu}\right)^2$$

$$\hat{K}_i^{opt} = \frac{w_i}{Nx_i},$$

$$w_{i+1} = w_i - N\hat{K}_i^{opt},\ x_{i+1} = 2 + \frac{N^2}{w_i}\left(\frac{\sigma}{\mu}\right)^2$$

Note that this iteration scheme only requires the knowledge of the coefficient of variation *cov* of the tasks' probability distribution ($cov = \sigma/\mu$). There are two extreme cases: 1. If $cov = 0$ (no variance) then this strategy assigns all jobs in a single round; 2. If $cov \to \infty$ (unbounded variance or negligible tasks' durations) then this scheme assigns jobs of size 1. (This scheme is not strictly factoring in the sense of Section 4 because the factor is not the same constant in subsequent rounds.)

5 Work stealing

So far we assumed that the master process cannot take back its decisions—i.e. once a job has been assigned to a worker, then the job must be processed by that worker. In the work stealing algorithm, the master process can reclaim already assigned but yet unprocessed tasks from the workers. The work stealing algorithm requires no a-priori information (not even the knowledge of latency). It initially assigns all the tasks in jobs of size W/N to idling worker processes. When a worker becomes idle again, the master reclaims all yet unprocessed tasks from all the worker processes and redistributes them equally back again to all worker processes. The periods between the redistributions are called rounds. Each round adds a penalty L' to the makespan.

An implementation of the work stealing algorithm can use two threads of control in each worker process: a "listening thread" which reacts to work redistribution messages by sending all yet unprocessed tasks to the master process; and a "working thread" which computes the tasks and notifies other processes when it runs out of work. These two threads share a queue of tasks. The queue is protected by a semaphore in order to exclude its simultaneous access by both the threads. The working thread repeats a loop in which it locks the queue, pops one task, unlocks the queue and starts processing the task. After finishing the task, this procedure repeats until the working thread finds the queue empty. Then it notifies the other processes and waits until the listening thread inserts tasks of the new round into the queue and resumes the computation (or terminates the whole process). *Yet unprocessed tasks* in a process are the tasks in the queue. A clever implementation of the algorithm amortises the latency by allowing a worker which reacts to a work redistribution message to continue in processing of tasks in its queue during the work redistribution.

As the task distribution in work stealing uses a more complex communication pattern (broadcasting and gathering) than the previous algorithms (point-to-point round-trip), we will denote this latency L', whereby $L' \ge L$. However, L' differs from L only by a constant factor if N is a constant. This factor depends on the physical mechanism which is used for communication among the processes. Note that $L' \approx L$ e.g. in a bus network or a network with a complete interconnection graph. Similarly as by factoring, there is no work imbalance at the end of the algorithm, therefore the competitive ratio of work stealing only depends on the number of rounds.

Theorem 3. *For all W, N, L, L' competitive ratio of the work stealing algorithm with no a-priori knowledge is $O((\ln W)/W)$ and approaches 1 if $W \to \infty$.*

Proof. The number of rounds of work stealing in the worst case can be determined as follows. Assume that one of the worker processes finishes its first job of size W/N, while no other worker process has finished its first task. After the redistribution a second round begins and the same situation happens: one worker process finishes its job, while none of the other worker processes has finished its first task. Etc. The total number of yet unprocessed tasks (in the whole system) is at most $w_i = W\left((N-1)/N\right)^i$ at the beginning of round i. At most N tasks are distributed at the beginning of the last round r. Solving $w_r \leq N$ for maximal r yields

$$r = \frac{\ln(W/N)}{\ln(N/(N-1))} \tag{4}$$

The rest of the proof is similar to the proof of Theorem 2. The competitive ratio of the work stealing algorithm can be bounded from above:

$$\mathcal{CR}_{\text{workstealing}}(W, N, L, L') \leq$$
$$\leq \frac{WT_{min}/N + T_{max} - T_{min} + L'\left(\ln(W/N)/\ln\frac{N}{(N-1)}\right)}{WT_{min}/N + L} =$$
$$= O((\ln W)/W) \tag{5}$$

This completes the proof. □

6 Comparison of deterministic assignment algorithms

It is clear from theorems 1, 2 and 3 that the chunking algorithm can not compete with the factoring and work stealing algorithms if the number of processes N is constant and the number of tasks W is sufficiently large in comparison with N ($W \approx N^3$ or larger).

We proved a common upper bound for competitive ratios of the work stealing and factoring algorithms for $W \to \infty$. Both these algorithms guarantee a perfect balance, therefore we can focus on their number of rounds which determine the cost of assignment. In order to keep things simple, we will assume $L = L'$ in the sequel. If we directly compare the number of rounds in work stealing (Eq. 4) and factoring (Eq. 2), then work stealing does not perform worse than factoring when $T \leq N + 1$, because then

$$\frac{r_{\text{workstealing}}}{r_{\text{factoring}}} = \frac{\ln\frac{N}{N-1}}{\ln\frac{1+T(N-1)}{(T-1)(N-1)}} \leq 1$$

However, the comparison above is not fair, because the work stealing algorithm with a-priori knowledge of $T = T_{max}/T_{min}$ is actually more efficient than in the proof of Theorem 3 (although it does not makes use of the knowledge of T). For a given T, let us reconsider the scenario in which always one worker process finishes all its tasks from the first round, while all the other worker processes do as little work as possible. While the worker computes its first job of size W/N tasks, all the other workers must have computed at least $W/(NT)$ tasks each. So every other worker has at most $W(T-1)/(NT)$ yet unprocessed tasks; in sum, there are at most $W(T-1)(N-1)/(NT)$ unprocessed tasks in the whole system (which is less than $W(N-1)/N$ tasks in the proof of Theorem 3). In the second round, each worker is assigned $W(T-1)(N-1)/\ (N^2T)$ tasks. When a worker finishes its job from the second round, then all the other workers have at most $W\left((T-1)/(NT)\right)^2$ yet unprocessed tasks each; in sum, there are at most $W\left((T-1)(N-1)/(NT)\right)^2$ yet unprocessed tasks in the whole system. Etc. Generally, there are $w_i = W\left((T-1)(N-1)/(NT)\right)^i$ yet unprocessed tasks in the whole system at the beginning of round i. At most N tasks are distributed at the beginning of the last round r. Solving $w_r < N$ for maximal r yields

$$r_{\text{workstealing}} = \frac{\ln(W/N)}{\ln\frac{NT}{(T-1)(N-1)}} \tag{6}$$

Fair comparison of the number of work stealing rounds (Eq. 6) with the number of factoring rounds (Eq. 2) yields

$$\frac{r_{\text{workstealing}}}{r_{\text{factoring}}} = \frac{\frac{\ln(W/N)}{\ln\frac{NT}{(T-1)(N-1)}}}{\frac{\ln(W/N)}{\ln\frac{1+T(N-1)}{(T-1)(N-1)}}} = \frac{\ln\frac{1+T(N-1)}{(T-1)(N-1)}}{\ln\frac{NT}{(T-1)(N-1)}} < 1$$

This means the work stealing algorithm performs better than the factoring algorithm for all W, N, L, L', T if L'=L. More precisely, the work stealing algorithm performs better for L, L' such that

$$\frac{\ln\frac{1+T(N-1)}{(T-1)(N-1)}}{\ln\frac{NT}{(T-1)(N-1)}} < \frac{L}{L'}$$

because then $L' r_{\text{workstealing}} \leq L r_{\text{factoring}}$. We stress that an a-priori knowledge of T is rarely available in practice and must therefore be estimated. With an inaccurate estimation of T, the factoring algorithm performs worse than in our analysis.

The work stealing algorithm is a clear winner. It has no parameters and requires no tuning. Moreover, it can be used (after some modifications) in applications where processes may fail or where the number of tasks may grow in run-time.

7 Conclusions

We analysed online performance of chunking, factoring and work stealing assignment algorithms in a deterministic model. The chunking algorithm requires an a-priori knowledge of the maximal task's duration and achieves competitive ratio N (which does not depend on W) for $W = \Omega(N^3)$, where N denotes the number of processes and W denotes the number of tasks. The performance of chunking algorithm is thus very poor, at least from the point of view of competitive analysis. The factoring algorithm requires an a-priori knowledge of the factor $T = T_{max}/T_{min}$. Its competitive ratio is bounded from above by $O(\ln(W)/W)$ and approaches 1 when $W \to \infty$, which is very desirable. The same holds for the deterministic work stealing algorithm, which performs better than the factoring algorithm and requires no a-priori information.

The last result is valid under two assumptions: 1. the underlying communication mechanism provides an efficient implementation of broadcasting and gathering, which we assume to be as fast as round-trip point-to-point communication; 2. the communication latency is constant which does not depend on the message size. The first assumption holds e.g. for bus and fully-switched networks; the second assumption holds for practically all contemporary networks, if the message size does not exceed a certain threshold.

References

1. I. Banicescu and S. Flynn-Hummel: *Balancing processor loads and exploiting data locality in irregular computations.* Technical Report RC 19934, IBM Research, 1995.
2. A. Borodin and R. El-Yaniv: *Online computation and competitive analysis.* Cambridge University Press, 1998.
3. R. Diekmann, A. Frommer, and B. Monien: *Efficient schemes for nearest neighbor load balancing.* Parallel Computing, **25** (7), 1999, 789–812.
4. R. Elsasser, A. Frommer, B. Monien, and R. Preis: *Optimal and alternating-direction loadbalancing schemes.* In: Proc. of Euro-Par, volume 1685 of Lecture Notes in Computer Science, Springer-Verlag, 1999, 280–290.
5. R. Elsasser, B. Monien, A. Frommer, and R. Preis: *Optimal diffusion schemes and load balancing on product graphs.* Parallel Processing Letters, **14** (1), 2004, 61–73.
6. L. E. Flynn and S. Flynn-Hummel: *Scheduling variable-length parallel subtasks.* Technical Report RC 15492, IBM Research, 1990.
7. S. Flynn-Hummel, E. Schonberg, and L. E. Flynn: *Factoring: A practical and robust method for scheduling parallel loops.* In: Proc. of Supercomputing '91, IEEE Computer Society / ACM, 1991, 610–619.
8. S. Fujita: *A semi-dynamic multiprocessor scheduling algorithm with an asymptotically optimal competitive ratio.* In: Proc. of the 8th International Euro-Par Conference on Parallel Processing, Springer-Verlag, 2002, 240–247.
9. M. R. Garey and D. S. Johnson: *Computers and intractability.* W. H. Freeman and Company, 1979.
10. T. Hagerup: *Allocating independent tasks to parallel processors: An experimental study.* Journal of Parallel and Distributed Computing, **47** (2), 1997, 185–197.
11. C. P. Kruskal and A Weiss: *Allocating independent subtasks on parallel processors.* IEEE Transactions on Software Engineering, **11** (10):1001–1016, 1985.
12. T. Plachetka: *Perfect load balancing for demand-driven parallel ray tracing.* In: B. Monien and R. Feldman, (eds), Proc. of Euro-Par 2002, volume 2400 of Lecture Notes in Computer Science, Springer-Verlag, 2002, 410–419.
13. T. Plachetka: *Tuning of algorithms for independent task placement in the context of demand-driven parallel ray tracing.* In: D. Bartz, B. Raffin, and H.W. Shen, (eds), Proc. of the Eurographics/ACM SIGGRAPH Symposium on Parallel Graphics and Visualization (EGPGV), Eurographics Proceedings Series, Eurographics Association, 2004, 101–109.
14. K. Pruhs, J Sgall, and E. Torng: *Online scheduling.* In: J.Y.-T. Leung, (ed.), Handbook of Scheduling, chapter 15, CRC Press, 2004, 15.1–15.41.