6.895 Theory of Parallel Systems Lecture 9

Analysis of Cilk Scheduler

Lecturer: Michael A. Bender *Scribe: Alexandru Caracaş, C. Scott Ananian*

Lecture Summary

1. *The Cilk Scheduler*
   We review the Cilk scheduler.

2. *Location of Shallowest Thread*
   We define the depth of a thread and the shallowest thread. Next, We prove that the shallowest thread on a processor is either at the top of a deque or being executed.

3. *Critical Threads*
   We construct a computation graph G' similar to the computation graph G, such that when a thread has no incomplete predecessors in G', then it is at the top of a deque.

4. *Execution Time Bounds For The* Cilk *Scheduler*
   We present the execution time bounds of the *Cilk* scheduler. We introduce an accounting argument to help in proving the bound.

5. *Performance Analysis of Scheduling Algorithm Using Delay Sequences*
   We analyze the performance of the *Cilk* work-stealing scheduling algorithm. We define delay sequences and use them to prove the execution time bounds of the *Cilk* scheduler.

1 The *Cilk* Scheduler

In this section we review the *work-stealing* algorithm of the *Cilk* scheduler. Recall that in *Cilk* a processor works on a procedure α until one of three events occur:

1. Procedure α *Spawns procedure* β. In this case the processor pushes α on (the bottom of) the ready deque, and starts work on procedure β.

2. Procedure α *Returns.* There are three cases:

   (a) If the deque is nonempty, the processor pops a procedure (from the bottom) and begins working on it.

   (b) If the deque is empty, the processor tries to execute α's parent (which is not in a deque).

   (c) If the deque is empty and the processor is unable to execute α's parent (because the parent is busy), then the processor work steals.

3. Procedure α *Syncs.* If there are no outstanding children, then continue: we're properly synced. Otherwise (when there are no outstanding children), work steal. Note that deque is empty in this case.

When a processor begins work stealing it operates as follows:

1. Processor chooses a victim *uniformly at random.*

2. If the victim's deque is empty, the processor repeats the first step.

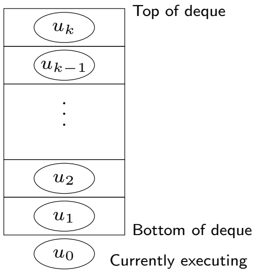


Figure 1: Deque of a processor.

3. Otherwise, the processor steals the top (oldest) thread from the victim's deque and begins to work on it.

Note: A thread can be in exactly one of three places:

1. in the deque,
2. on a processor being executed, or
3. waiting for predecessors to complete.

In the following we use the notion of ***procedure*** as composed of multiple threads.

2 Location of shallowest thread

In this section we define the depth of a thread and the shallowest thread. Next we prove that the shallowest thread on a processor is either at the top of a deque or being executed.

Definition 1 (Depth of thread) *The **depth** $d(u_i)$ of a thread u_i is the longest distance in the DAG from the root thread to u_i.*

Definition 2 (Shallowest Thread) *The **shallowest thread** is the thread with minimal depth.*

We prove some structural properties of the deque.

Lemma 3 (Structural lemma) *Consider any processor p at time t. Let u_0 be the current root thread (of procedure α_0) executing on processor p. Let $u_1, u_2, \ldots, u_k$ be the threads (of procedures $\alpha_1, \alpha_2, \ldots, \alpha_k$) in p's deque ordered from bottom to top. Let $d(u_i)$ be the depth of thread u_i in DAG G. Then,*

$$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

Observation 4 *A procedure is made up of multiple threads, but* at a given time t, *there is only one thread in the procedure which is being executed.*

Corollary 5 *The shallowest thread in a processor is either at the top of a deque or being executed.*

Proof Proof is by *induction on actions.* The **base case** (deque is empty; u_0 executing) trivially satisfies the inequality. Thereafter, only four possible actions can alter the currently-executing thread and/or the deque. We prove that each of these actions maintains the desired depth-ordering property.

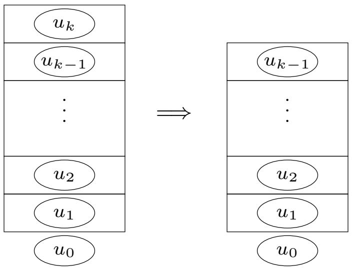


Figure 2: Change in the deque of a processor when a procedure is stolen.

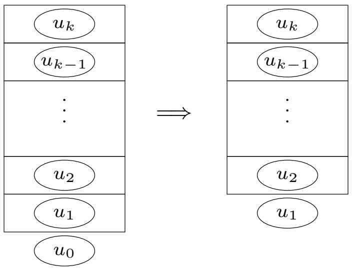


Figure 3: Change in the deque of a processor when the processor returns from a procedure.

- **Case 1: Steal.** A steal removes the top entry from the deque (see Figure 2). The processor performing the steal begins executing u_k with an empty deque, trivially satisfying the inequality as in the base case. The processor from whom the thread was stolen also maintains the invariant, because if before the steal
$$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k),$$
after the steal
$$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}).$$
Removing threads from anywhere in the deque cannot invalidate the ordering.

- **Case 2: Return.** A return removes the bottom entry from the deque (see Figure 3). As in the case of the steal, removing terms from the inequality cannot invalidate it. If before the return
$$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k),$$
afterwards we have
$$d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$
As before, removing threads from anywhere in the deque cannot invalidate the ordering.

- **Case 3: Continue or sync.** A sync or continue replaces the bottom entry of the deque (see Figure 4) with a thread with depth increased by one. A sync with one or more children outstanding becomes a steal, and is handled by Case 1. If all children are complete, then a sync and a continue are identical: we replace the currently-executing thread u_0 with its successor in the DAG, u_a. Figure 5 shows the relevant piece of the computation DAG.

  If the longest path to u_0 has length $d(u_0)$, then clearly there is a path to u_a through u_0 which has length $d(u_0) + 1$. Therefore,
$$d(u_a) \geq d(u_0) + 1,$$

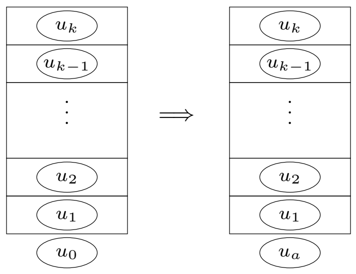


Figure 4: Change in the deque of a processor when the processor reaches a sync point or continues a procedure.

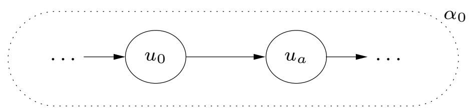


Figure 5: A piece of a computation DAG. Thread u_0 is followed by thread u_a in procedure α_0.

and since $d(u_0) \geq d(u_1)$, then

$$d(u_a) > d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

- **Case 4: Spawn.** A spawn replaces the bottom entry in the deque and also pushes a new procedure (see Figure 6). Figure 7 shows the portion of the computation DAG involved in a spawn. Note that, since the only edges to u_a and u_s come from u_0, that u_a and u_s have the same depth in the DAG, namely $d(u_0) + 1$. Therefore since $d(u_0) \geq d(u_1)$, then $d(u_s) = d(u_a) > d(u_1)$ and, as required

$$d(u_s) \geq d(u_a) > d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

  Note: This is the only case where the depth of the currently-executing thread may become equal to the depth of the thread on the bottom of the deque; hence the only case where a shallowest thread may be the currently-executing thread instead of the thread at the top of the deque.

Since the base case satisfies the property, and every action maintains the property, then at any time t in the execution for all deques

$$d(u_0) \geq d(u_1) > \cdots > d(u_{k-1}) > d(u_k).$$

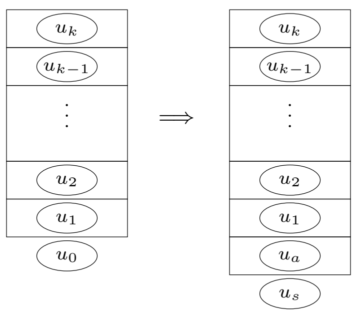


Figure 6: Change in the deque of a processor when the processor spwans a procedure.

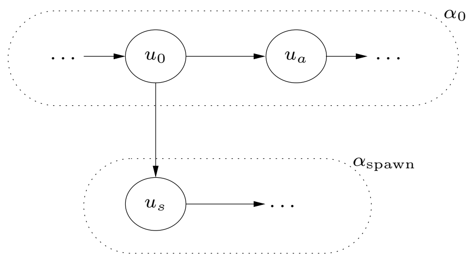


Figure 7: A piece of a computation DAG. Thread u_0 is followed by thread u_a in procedure α_0, and spawns thread u_s in procedure α_{spawn}.

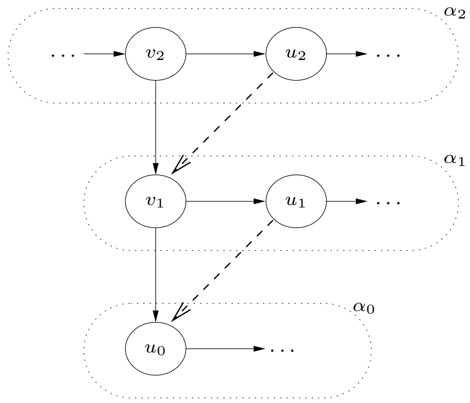


Figure 8: An augmented DAG corresponding to the execution DAG of Figure 9. The dashed edges have been added from continuation threads to spawn threads.

□

3 Critical threads

In this section we construct an augmented computation DAG G' similar to the computation DAG G, such that when a thread has no incomplete predecessors in G', then it is at the top of a deque. Next, we introduce **critical threads**.

Our goal is to show that any thread on the top of a deque cannot stay there long because it will be stolen.

Definition 6 (Construction of $G\prime$) *We create an augmented DAG $G\prime$ by making a (back) edge to every spawn thread from the parent's continuation thread. Note that if a sync is immediately followed the spawn, we need to insert an extra continuation thread before the sync to prevent cycles in the DAG. Figure 8 shows the execution graph of Figure 9 augmented with these edges.*

We show that **work-stealing** makes progress on the **critical path**. Roughly speaking, $\Theta(P)$ steals decrease the critical path by a constant. We require a stronger property for the computation DAG.

Proposition 7 (Necessary property) *Whenever a thread has no incomplete predecessors in the augmented DAG $G\prime$, it is being executed or is on the top of some deque.*

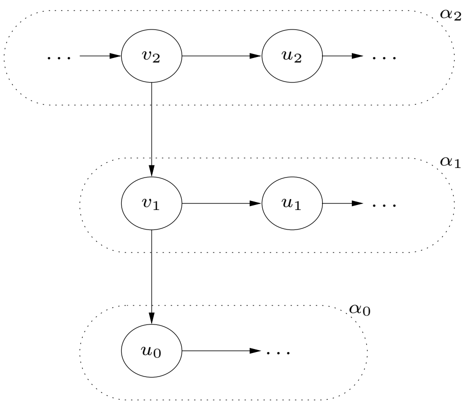
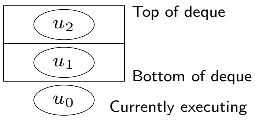


Figure 9: An execution DAG with threads v_2 and u_2 in procedure α_2, threads v_1 and u_1 in procedure α_1, and thread u_0 in procedure α_0.

Note: Proposition 7 does not hold for the execution DAG G, that is why we create G'.

Example 1 To see why, consider the execution DAG in Figure 9. After threads v_2 and v_1 have been executed, we begin to execute thread u_0. We have the following deque:



The thread u_1 has no unexecuted precedessors left in the DAG, but it is neither being executed nor at the top of the deque.

Observation 8 (Depth of Augmented DAG G') *The critical path of the augmented DAS G' is $2T_\infty$. We can now get to a spawned thread via our back edge from the continuation edge, adding a distance of one to the longest path to the spawned thread. But we can only add one such extra edge to the path per spawn edge on the critical path of the original graph G. If we define the* ***depth of a graph*** *$D(G)$ to be the depth of the deepest node, then,*

$$D(G) \leq D(G') \leq 2D(G),$$

where $D(G) = T_\infty$.

Proposition 7 guarantees that whenever you execute a thread, you will be executing one of the **shallowest** threads in the augmented DAG G'. Hence, when you steal work (i.e. not otherwise making progress) you make progress on the critical path by executing the threads which are shallowest.

Definition 9 (Critical thread) *A thread is* **critical** *if it has no un-executed predecessors in the augmented DAG G'.*

Note: The extra edges in the augmented DAG G' are not execution edges, they simply induce extra ordering that we require for our performance analysis of the *Cilk* scheduler.

4 Execution Time Bounds For The *Cilk* Scheduler

The *Cilk* work-stealing scheduling algorithm has provably good execution time bounds. We introduce an accounting argument based on **two buckets** to help in proving the bound.

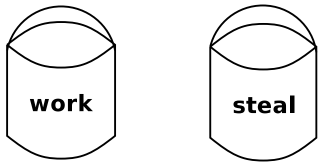


Figure 10: The **work** and **steal** buckets used in the accounting argument.

Theorem 10 (Cilk Scheduler execution time bound) *Consider the execution of any fully strict multithreaded computation with work T_1 and critical path T_∞ by* Cilk*'s work-stealing algorithm on a parallel computer with P processors. For any $\varepsilon > 0$, with probability at least $1-\varepsilon$, the execution time on P processors is*

$$T_P \leq \frac{T_1}{P} + O\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right). \tag{1}$$

Proof Idea The proof is based on an **accounting argument**. We have 2 buckets, one for **work** and one for **steal**. On each time-step, a processor puts one dollar into one of the two buckets, depending on the action that it performs in that step. The possible two actions and their respective results are:

- execute instruction – one dollar into **work** bucket,
- steal attempt – one dollar into **steal** bucket.

We assume that all threads are of equal size and that all processors have the same time-step. In the following We overload the notation of steal attempt with steal.

□

Lemma 11 *At the end of computation, the number of dollars in the* **work** *bucket is T_1.*

Proof The total work is exactly T_1 so at the end of the computation exactly T_1 work has been executed. Thus the number of dollars in the **work** bucket is precisely T_1. □

We know bound the steals with the following lemma.

Lemma 12 (Scheduler) *At the end of a computation the number of dollars in the* **steal** *bucket is $O(PT_\infty)$, with* **high probability**.

We give an exact definition of **high probability** during the course of the proof. In the rest of the lecture, we prove the previous lemma. We introduce several concepts to prove the bound.

Observation 13 (Dollars) *Per time-step P dollars enter the buckets.*

Thus, the running time on P processors is the amount of time it takes to fill out all the buckets.

Claim 14 *The running time on P processors is less than the total number of dollars (actions) divided by the number of processors:*

$$\begin{aligned} T_P &\leq \frac{\#\text{dollars in work bucket} + \#\text{dollars in steal bucket}}{P} \\ &\leq \frac{T_1}{P} + O(T_\infty). \end{aligned} \tag{2}$$

We know how many dollars will be in the work bucket. We need a bound for the number of dollars in the steal bucket.

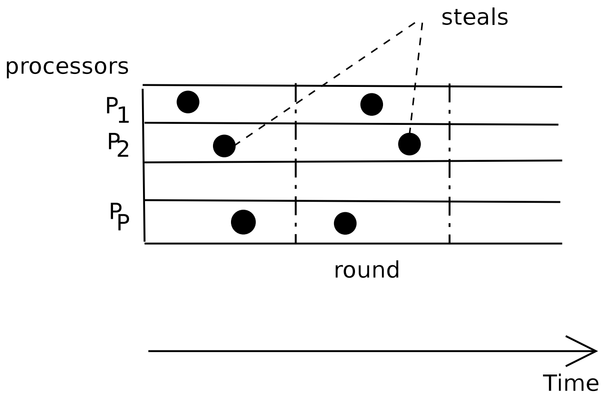


Figure 11: Dividing computation into steal rounds.

5 Performance Analysis of Scheduling Algorithm Using Delay Sequences

In this section, we analyze the performance of the *Cilk* work-stealing scheduling algorithm. We define ***delay sequences*** and use them to prove the execution time bounds of the *Cilk* scheduler.

We prove equation (2) in the rest of the lecture.

Observation 15 *There is no constant in front of the term T_1/P in the previous bound. All scheduling overhead cost is amortized against the critical path T_∞. Also, one important observation is that the term T_1/P is much bigger than the term T_∞.*

5.1 Rounds

As long as no processor is stealing, the computation is progressing efficiently. What divide the computation into **rounds**, which (more or less) contain the same number of steals (see Figure 11).

Definition 16 (Round) *A* ***round*** *of work stealing attempts is a set of at least P and at most $2P-1$ steal attempts. We assume that steal attempts complete in one unit of time.*

Round 1 begins at time 0 and ends when there have been at least P steals. In general, round i begins when round $i-1$ ends, and round i ends when there have been at least P steals.

Recall by Definition 9 that a **critical thread** is at the top of some deque or is being executed.

Proof Idea The main idea is that, if a thread is on the top of a deque for CP steal attempts, then the probability that it would executed for C rounds is very small, namely:

$$\left(1 - \frac{1}{P}\right)^{CP} \leq e^{-C}.$$

□

The idea is that a thread is unlikely to be critical for many rounds. Also, a thread may be ready for many rounds but not critical.

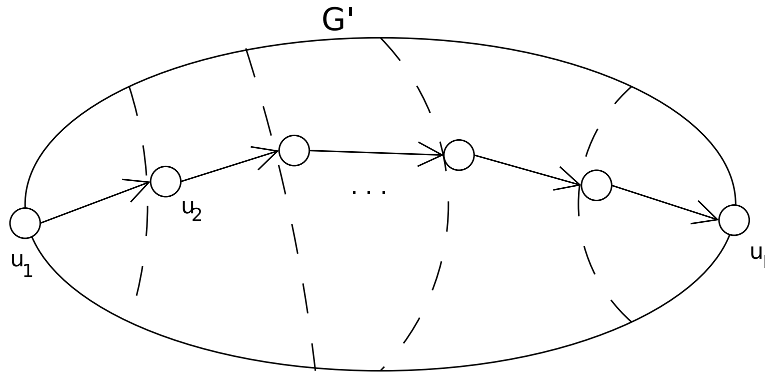


Figure 12: A directed path $U = (u_1, u_2, \ldots, u_L)$ in the augmented DAG $G\prime$.

Theorem 17 *The probability that a critical thread is un-stolen for C rounds is:*

$$\left(1 - \frac{1}{P}\right)^{CP} \leq e^{-C}. \tag{3}$$

Proof For C rounds, there are at least CP steals. The steal process is done at random and there are P places that a processor could steal from. Hence, the probability that a critical thread would be stolen during a round is $1/P$, and the probability that the critical thread will not be stolen is $1 - 1/P$. Hence, for C rounds we obtain the probability that a critical thread is not stolen:

$$\left(1 - \frac{1}{P}\right)^{CP}.$$

For a simplified bound of the above equation we use Euler's inequality:

$$\left(1 + \frac{1}{x}\right)^{x} \leq e.$$

We obtain

$$\left(1 - \frac{1}{P}\right)^{CP} \leq e^{-C}.$$

□

For a small number of rounds a thread might not be executed. However, for a large number of rounds a thread will be executed with very high probability, because the term e^{-C} in the previous bound becomes exponentially small. If a thread is at the top of a deck then it has no un-executed predecessors in the augmented DAG $G\prime$. Intuitively we would expect that in every time-step we are able to peal some threads from the augmented DAG such that in time $O(PT_\infty)$ there would be no augment DAG left (see Figure 12). This idea is expressed in the following lemma and explanation.

5.2 Delay Sequence Argument

Lemma 18 (Structural Lemma) *There exists a directed path $U = (u_1, u_2, \ldots, u_L)$ in G' in which, during every time-step of the computation, some thread in U is critical.*

Proof The proof is by induction backwards in time.

Base case: During last time-step t, last thread in G' is u_L and it is critical and executed (see Figure 12).

Induction step: Suppose during time-step t, thread u_j is critical. During time-step $t-1$, either u_j is also critical, or if not it is because it has un-executed predecessors in $G\prime$. Some threads $u_i, u_{i}\prime, \ldots \prec_{G\prime} u_j$ are being executed during t-1. One of them is minimal and is critical.

Main idea: We prove that since the probability that any thread remains critical for a long period of time is small, there cannot be too many steal attempts. □

Observation 19 *An executing thread is not necessarily critical. The very last thread in the computation u_L when it is being executed it is critical. The extra edges in the augmented DAG introduce additional ordering (see Figure 9).*

In order to complete the proof for the execution time bound of the *Cilk* scheduler, we introduce some definitions and lemmas.

Definition 20 (Delay Sequence) *A* ***delay sequence*** *is an event described by the triple (U, R, Π) where:*

- $U = (u_1, u_2, \ldots, u_L)$ *is a directed path in G' from* **source** *of DAG to* **sink**,
- $R > 0$ *is an integer,*
- $\Pi = (\pi_1, \pi_2, \ldots, \pi_L)$ *is a partition of R, namely $\sum_i \pi_i = R$*

A ***delay sequence*** *occurs if $\forall i$ at least π_i rounds occur while u_i is critical and un-executed.*

A ***delay sequence*** is a combinatorial object. A ***delay sequence*** occurs if there is a directed path in $G\prime$ in which one of the threads in the path is critical at each time-step.

Lemma 21 *If at least $2P(2T_\infty + R)$ steal attempts occur, then some (U, R, Π) delay sequence occurs.*

Proof Consider a path $U = (u_1, u_2, \ldots, u_L)$ for which during each time-step t some thread is critical. There are at most $2T_\infty$ rounds during which a thread in U executes, since $L \le 2T_\infty$. For all the other R rounds, some thread is critical and not executed.
Note: The converse need not be true. □

The idea is to sum up over all possible events the probability that the event occurs. Show that the sum is small. We would then be able to show that with *high probability* there at most $2P(2T_\infty + R)$ steal attempts.

A ***delay sequence*** is an **event** that explains why there are at least $2P(2T_\infty + R)$ steal attempts. The probability that at least $2P(2T_\infty + R)$ steal attempts occur is at most the probability that there exists a delay sequence (U, R, Π) that occurs.

Observation 22 (Number of Steal Attempts) *We make the following observation about the probability that at least $2P(2T_\infty + R)$ steal attempts occur.*

$$\begin{aligned} Pr\left[\begin{array}{c}\text{at least } 2P(2T_\infty + R)\\ \text{steal attempts occur}\end{array}\right] &\le Pr\left[\begin{array}{c}\text{exists delay sequence}\\ (U,R,\Pi)\text{ that occurs}\end{array}\right] \\ &\le \sum_{\substack{\text{delay sequences}\\ (U,R,\Pi)}} Pr\left[(U,R,\Pi)\text{ occurs}\right] \\ &\le \left(\begin{array}{c}\text{Number of}\\ \text{Delay Sequences}\end{array}\right)\left(\begin{array}{c}\text{Maximum Probability}\\ \text{that any delay}\\ \text{sequence occurs}\end{array}\right). \end{aligned}$$

Lemma 23 (Probability of Delay Sequence) *The probability that a particular delay sequence (U, R, Π) occurs is e^{-R}, which is exponentially small in the number of rounds, R.*

Proof

$$\begin{aligned}
\Pr\left[(U,R,\Pi)\,\text{occurs}\right] &= \prod_{1\le i\le L} \Pr\left[\begin{array}{c}\pi_i\,\text{rounds occur}\\ \text{while } u_i \text{ is critical}\end{array}\right]\\
&\le \prod_{1\le i\le L} e^{-\pi_i}\\
&= e^{-R}.
\end{aligned}$$

□

Lemma 24 (Number of Delay Sequences) *The number of delay sequences is at most*

$$2^{2T_\infty}\binom{2T_\infty + R}{2T_\infty}.$$

Proof We have

$$\#\text{paths} \le 2^{2T_\infty},$$

because no path is longer that $2T_\infty$ and the out degree of any thread (vertex) is at most two. Thus, there are two choices (either a continuation edge or a spawn edge) starting from the root node. Also

$$\begin{array}{c}\#\text{partitions of a path}\\ \text{in augmented DAG } G\prime\end{array} = \binom{2T_\infty + R}{2T_\infty}.$$

We are partitioning R into L pieces. However, for large R, this is smaller than partitioning R into $2T_\infty$ pieces. We simply count up all the possible ways of partitioning the path. Thus, by multiplying the number of paths with the number of ways in which a path can be partitioned we obtain the number of delay sequences, which is at most

$$2^{2T_\infty}\binom{2T_\infty + R}{2T_\infty}.$$

□

Our goal is to show that the probability that any delay sequence occurs is very small. Since we know the probability of a given delay sequence we use the *Boole's inequality* to compute the probability of any delay sequence. Recall that:

$$\Pr\left[A \vee B \vee C \vee \cdots\right] \le \Pr\left[A\right] + \Pr\left[B\right] + \Pr\left[C\right] + \cdots.$$

Following from Lemma 23 and Lemma 24 we have that:

$$\begin{aligned}
\Pr\left[\text{any } (U,R,\Pi)\,\text{occurs}\right] &\le \left(\begin{array}{c}\text{Number of}\\ \text{Delay Sequences}\end{array}\right)\left(\begin{array}{c}\text{Maximum Probability}\\ \text{that any delay}\\ \text{sequence occurs}\end{array}\right)\\
&\le 2^{2T_\infty}\binom{2T_\infty + R}{2T_\infty} e^{-R}
\end{aligned}$$

Next, we use the following "death-bed" formula to simplify the above inequality.

$$\left(\frac{y}{x}\right)^x \le \binom{y}{x} \le \left(\frac{ey}{x}\right)^x.$$

We have:

$$\begin{aligned}\Pr\left[\text{any } (U,R,\Pi)\,\text{occurs}\right] &\le 2^{2T_\infty}\left(\frac{e(2T_\infty+R)}{2T_\infty}\right)^{2T_\infty} e^{-R} \\ &\le \left(\frac{2e(2T_\infty+R)}{2T_\infty}\right)^{2T_\infty} e^{-R}\end{aligned}$$

In the right part of the inequality, the first term is considerably large, while the second term is exponentially small in R. We want to choose a value for R such that the product of the two terms is small. We let $R = 2CT_\infty$ where C is a constant. Replacing in the previous equation we have:

$$\Pr\left[\text{any } (U,R,\Pi)\,\text{occurs}\right] \le \left(\frac{[2e(C+1)]^{1/C}}{e}\right)^{R}. \tag{4}$$

Definition 25 (High Probability) *An event E occurs with **high probability** if the probability that the event occurs that is $Pr[E] \ge 1 - P(n)$, where $P(n)$ is a polynomial. That is the probability that the event does not occur is polynomially small.*

Observation 26 *When $C \ge 4$, probability decreases exponentially in R.*

Example 2 If we take $C = 4$ then the probability is less or equal to 0.84^R.

Let:

$$\varepsilon = \left(\frac{[2e(1+C)]^{1/C}}{e}\right)^{R}.$$

Then $\Pr\left[\begin{array}{c}\ge 2P(2T_\infty+R)\\ \text{steal attempts occur}\end{array}\right] \le \varepsilon$, when $R = \Omega(\lg\left(\frac{1}{\varepsilon}\right))$.

6 Wrap-Up

We have shown that with **high probability** the number of steal attempts is small. We have come up with an event which explains why there are steal attempts. This event is a **delay sequence**. Next we counted the number of events. We computed the probability that such an event occurs. By summing up all the possible events we have shown that with **high probability** there are few steal attempts. With probability at least $1-\varepsilon$ the number of steal attempts is $O(P\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right))$.

We have started our analysis using an **accounting argument**, based on a **work** bucket and a **steal** bucket.We have shown that at the end of the computation there are T_1 dollars in the **work** bucket and with probability at least $1-\varepsilon$ there are $O(P\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right))$ dollars in the **steal** bucket. Thus, with probability at least $1-\varepsilon$:

$$T_P \le \frac{T_1}{P} + O\left(T_\infty + \lg\left(\frac{1}{\varepsilon}\right)\right).$$

6.1 Death-Bed Formulae

Figure 13 shows a list of useful ("death-bed") formulae that were used in the analysis of the *Cilk* scheduler.

Euler's inequality:

$$\left(1+\frac{1}{x}\right)^{x} \leq e \leq \left(1+\frac{1}{x}\right)^{x+1}, \tag{5}$$

$$\left(1-\frac{1}{x}\right)^{x} \leq \frac{1}{e} \leq \left(1-\frac{1}{x}\right)^{x-1}. \tag{6}$$

Boole's inequality (Union bound):

$$\Pr\left[A \vee B \vee C \vee \ldots\right] \leq \Pr\left[A\right] + \Pr\left[B\right] + \Pr\left[C\right] + \ldots. \tag{7}$$

Combination of y choose x bound:

$$\left(\frac{y}{x}\right)^{x} \leq \binom{y}{x} \leq \left(\frac{ey}{x}\right)^{x}. \tag{8}$$

Figure 13: Death-bed formulae.