Critical path estimation in heterogeneous scheduling heuristics

Thomas McSweeney*

2nd September 2020

1 Introduction

The *critical path* of a project is defined as the longest sequence of constituent tasks that must be done in order to complete it [9, 8]. If we express the project in the form of a task graph, then the critical path is simply the longest—i.e., costliest—path through it. This is useful because the time it takes to execute the tasks on the critical path of the graph is therefore a lower bound on the makespan of any possible schedule for the project, no matter how many processors are available. In the context of a listing heuristic for scheduling the task graph on a set of parallel processors, a natural choice then is to prioritize all tasks according to the length of the critical path from that task to the end, the idea being that tasks with the greatest downward path length contribute most heavily to the makespan and should therefore be processed as soon as possible. This approach has a long and successful history in scheduling for homogeneous processors, and is in fact provably optimal for two-processor systems [4].

Now, in the homogeneous case, all tasks have the same cost on all processors, so that there is only one possible cost each task may take. In particular this means that the node weights in the task graph are *fixed* so, disregarding the edge weights for the moment, the longest path through the DAG is also fixed, no matter what schedule we ultimately follow. Therefore, when we refer to the critical path of the task graph it is clear what is meant. Unfortunately, the concept of the critical path is not so clearly defined for heterogeneous processing environments. The problem is, there are now multiple possible costs that each task may take—depending on the schedule—and likewise many different communication costs that may be incurred. In particular this means that the weights of the task DAG are not fixed and we cannot simply compute a longest path. Consider

*School of Mathematics, University of Manchester, Manchester, M13 9PL, England (thomas.mcsweeney@postgrad.manchester.ac.uk).

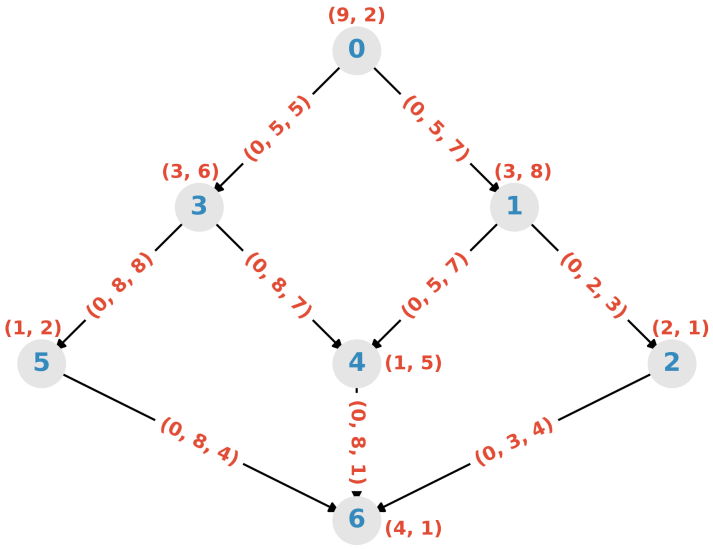


Figure 1: Simple task graph with costs on a two-processor target platform.

for example the simple DAG shown in Figure 1, where the labels represent all the possible weights each task/edge may take on a two-processor heterogeneous target platform; the red labels near the nodes represent the computation costs on processors $P1$ and $P2$ in the form (W_i^1, W_i^2), while the edge labels represent the possible communication costs in the form $(W_{ik}^{11} = W_{ik}^{22} = 0, W_{ik}^{12}, W_{ik}^{21})$. How should the longest path through a graph like that in Figure 1 be defined?

The HEFT approach, as described in previous chapters, is to use *average values* over all sets of possible costs in order to fix the DAG weights and then compute the critical path in a standard dynamic programming manner. In particular, for each $i = 1, \ldots, n$, we compute a number u_i, called the upward rank, which purportedly represents the critical path length from task t_i to the end and is taken to be the task's priority. For example, for the graph in Figure 1, HEFT first (implicitly) converts the DAG to the fixed-cost one shown in Figure 2, and then calculates the task ranks like so:

$$
\begin{aligned}
u_6 &= 2.5, \\
u_5 &= 1.5 + 3 + 2.5 \\
&= 7, \\
u_4 &= 3 + 2.25 + 2.5 \\
&= 7.75, \\
u_2 &= 1.5 + 1.75 + 2.5 \\
&= 5.75, \\
u_3 &= 4.5 + \max\{4 + u_5,\ 3.75 + u_4\}
\end{aligned}
$$

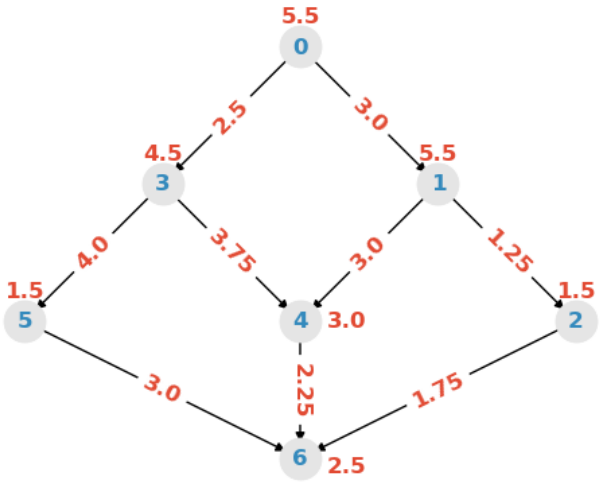


Figure 2: Fixed-cost counterpart of task DAG from Figure 1 which is implicitly used in HEFT.

$$
\begin{aligned}
&= 4.5 + \max\{11,\ 11.5\} \\
&= 16, \\
u_1 &= 5.5 + \max\{3 + u_4,\ 1.25 + u_2\} \\
&= 5.5 + \max\{10.75,\ 7\} \\
&= 16.25, \\
u_0 &= 5.5 + \max\{2.5 + u_3,\ 3 + u_1\} \\
&= 5.5 + \max\{18.25,\ 19.25\} \\
&= 24.75.
\end{aligned}
$$

These ranks give a scheduling priority list of $\{t_0, t_1, t_3, t_4, t_5, t_2, t_6\}$. The resulting schedule length obtained by HEFT with these priorities is 22, as shown in Figure 3. Interestingly, we see that u_0, the rank of the single entry task, is greater than the schedule makespan—and therefore obviously not a lower bound on it. The question then is, what quantity do the u_i values actually represent? Now, because of the averaging of the costs, perhaps the most intuitive interpretation is that the u_i are in turn estimates of the average critical path lengths, taken over the set of all possible critical paths —but there is no robust mathematical justification for believing that this definition of the critical path is more useful than other possibilities.

Given this ambiguity, alternative ways to define the critical path for the ranking phase in HEFT have been considered before, most notably by Zhao and Sakellariou [14], who empirically compared the performance of HEFT when averages other than the mean (e.g., median, maximum, minimum) are used to compute upward (or downward) ranks. Their conclusions were that using the

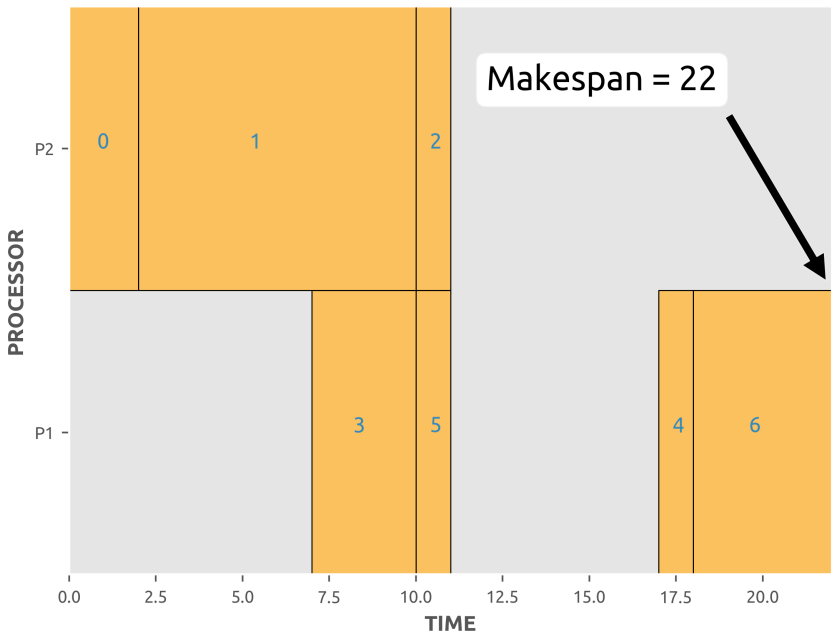


Figure 3: HEFT schedule for DAG in Figure 1.

mean is not clearly superior to other averages, although none of the other options they considered were consistently better. Indeed, perhaps the biggest takeaway from their investigation was that HEFT is very sensitive to how priorities are computed, with significant variation being seen for different graphs and target platforms. In this chapter we undertake a similar investigation with the aim of establishing if there are choices which do consistently outperform the standard upward ranking in HEFT. In addition, we consider how critical path estimates can be used to determine processor selection, as well as task prioritization, following the approach of the *Predict Earliest Finish Time* (PEFT) heuristic [1], and attempt to ascertain which definition leads to the best practical performance.

This will be an empirically-driven study, as is common in this area. To facilitate this investigation we created a software package that simulates heterogeneous scheduling problems, much like that described in the previous chapter, although not restricted to accelerated target platforms. As before, the (`Python`) source code for this simulator can be found on Github[1] and all of the results presented here can be re-run from scripts contained therein.

[1] https://github.com/mcsweeney90/critical-path-estimation

2 A universal lower bound

Functionally, the critical path is used in HEFT as a lower bound on the makespan, so that minimizing the critical path gives us the most scope to minimize the makespan, assuming we make good use of our parallel resources. With this in mind, there are many different ways we can define the critical path so that it gives a lower bound on the makespan of any possible schedule. The most straightforward approach would be to just set all weights to their minimal values but a tighter bound can be computed in the following manner. First, define ℓ_i^a for all tasks t_i and processors p_a to be the critical path length from t_i to the end (inclusive), assuming that it is scheduled on processor p_a. These values can easily be computed recursively by setting $\ell_i^a = W_i^a$, $a = 1, \ldots, q$, for all exit tasks then moving up the DAG and calculating

$$\ell_i^a = W_i^a + \max_{k \in S_i} \left(\min_{b=1,\ldots,q} \left(\ell_k^b + W_{ik}^{ab} \right) \right), \quad a = 1, \ldots, q, \tag{1}$$

for all other tasks. Then, for each $i = 1, \ldots, n$,

$$\ell_i = \min_{a=1,\ldots,q} \ell_i^a \tag{2}$$

gives a true lower bound on the remaining cost of any schedule once the execution of task t_i begins. These ℓ_i values could be useful as alternative task priorities in HEFT, especially since the cost of computing all of the ℓ_i^a in this manner is only $O((m+n)q) \approx O(n^2 q)$ so in particular is the same order as the usual HEFT prioritization phase.

For example, for the simple DAG shown in Figure 1, we find that the ℓ_i values are as given in Table 1 (with the u_i included for comparison). Interestingly, we see that tasks t_1 and t_3 have the same lower bound (8 units) and the performance of the alternative ranks relative to the standard u_i sequence in HEFT depends on which is chosen to be scheduled first: if $t-1$, the priority list does not change so the schedule makespan is 22 units, but if t_3 is selected instead, the final schedule makespan is 20 units—i.e., smaller than the original—as illustrated in Figure 4. Of course, this is only one example: there is no mathematically valid reason to suppose that using the ℓ_i sequence instead of u_i as the task ranks in HEFT will actually lead to superior performance in general. Still, it seems worthwhile to investigate this empirically, which we do in Section 5.

(Note that the lower bound on the critical path as defined here is very similar to the optimistic cost used in PEFT; this will be discussed further in Section 4.)

3 A stochastic interpretation

In this section we propose a family of alternative task ranking phases in HEFT based on the following interpretation of the standard ranking. Given the complex

Table 1: Upward ranks u_i and lower bounds ℓ_i for the DAG in Figure 1.

Task:	0	1	2	3	4	5	6
u_i	24.75	16.25	5.75	16	7.75	7	2.5
ℓ_i	16	8	2	8	5	3	1

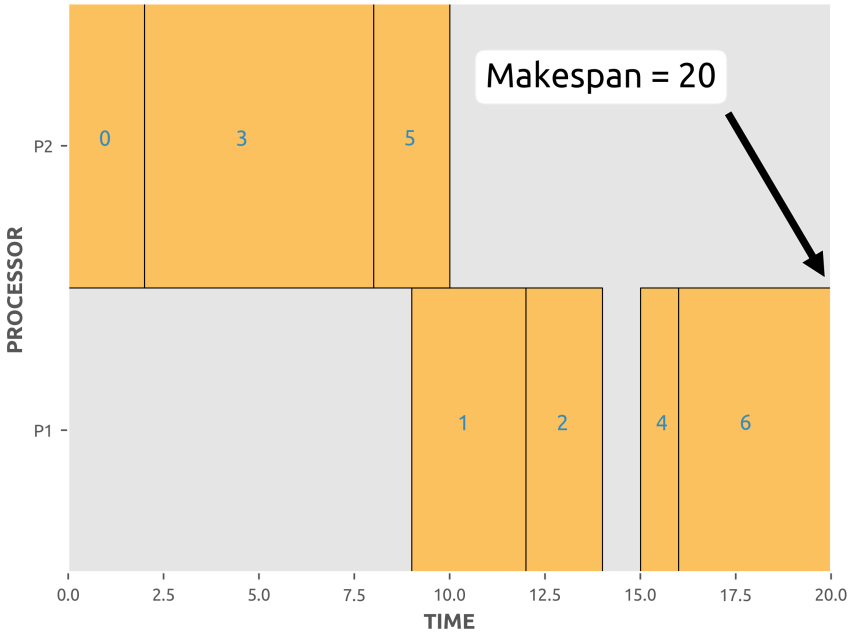


Figure 4: Alternative HEFT schedule for DAG in Figure 1.

interplay between the ranking and processor selection phases, it is impossible to predict exactly which values all DAG weights will take at runtime, at least without modifying the latter. Now, by using average values over all sets of possible costs, HEFT in some sense implicitly assumes that all members of each set are equally likely to be incurred. Conceptually, we can view this as an attempt to *model* the processor selection phase in order to predict which values the weights will assume. This model takes the form of a *stochastic graph*—i.e., all weights are assumed to be independent (discrete) random variables with associated probability mass functions (pmfs) given by the assumption of equal likelihood. More precisely, let m_i be the pmf corresponding to the task weight variable w_i and m_{ik} that for the edge weight w_{ik}, then

$$m_i(W_i^a) \coloneqq \mathbb{P}[w_i = W_i^a] = \frac{1}{q}, \quad a = 1, \ldots, q,$$

and

$$\begin{aligned} m_{ik}(W_{ik}^{ab}) &= m_i(W_i^a) \cdot m_k(W_k^b) \\ &= \frac{1}{q^2}, \quad \forall a, b. \end{aligned}$$

It is important to note here that the model defined by these pmfs is clearly not accurate. In particular, all of the node and edge weight variables are independent of one another. This is induced by the averaging but does not reflect, for example, the fact that edge weights are fully determined once the node weights are realized. Still, as the saying goes, all models are wrong, some are just more useful than others; we attempt to establish exactly how useful this one is through extensive numerical simulations in Section 5.

Note also that the expected values of the node and edge weight variables are given by

$$\mathbb{E}[w_i] = \sum_{a=1}^{q} W_i^a m_i(W_i^a) = \frac{1}{q} \sum_{a=1}^{q} W_i^a, \tag{3}$$

$$\mathbb{E}[w_{ik}] = \sum_{a=1}^{q} \sum_{b=1}^{q} W_{ik}^{ab} m_{ik}(W_{ik}^{ab}) = \frac{1}{q^2} \sum_{a,b} W_{ik}^{ab}. \tag{4}$$

This means that $\mathbb{E}[w_i] = \overline{w_i}$ and $\mathbb{E}[w_{ik}] = \overline{w_{ik}}$. So the computation of the upward ranks u_i in HEFT can instead be done by setting $u_i = \mathbb{E}[w_i]$ for all exit tasks, then moving up the DAG and recursively computing

$$u_i = \mathbb{E}[w_i] + \max_{k \in S_i} \big(u_k + \mathbb{E}[w_{ik}] \big) \tag{5}$$

for all other tasks.

In summary, since all possible node and edge weights of a task graph G are known but their actual values at runtime aren't, one possible interpretation of the standard HEFT task prioritization phase is that critical path lengths in G are estimated through a two-step process:

1. An associated stochastic graph G_s is implicitly constructed with node and edge pmfs m_i and m_{ik} as defined above.

2. The numbers u_i are recursively computed for all tasks in G_s using (5), and taken as the critical path lengths from the corresponding tasks in G.

In the following two sections, we propose modifications of both steps so as to obtain different critical path estimates that may be used as task ranks in HEFT. The performance of these will then be evaluated through extensive numerical simulations in Section 5.

3.1 The critical path of G_s

Now, since all of its weights are RVs, the critical path of the stochastic graph G_s is clearly itself a random variable. But a natural question arises from the interpretation outlined in the previous section: what is the relationship between the sequence of numbers u_i as defined by (5) and the critical path of G_s? In fact, it has long been known in the context of *Program Evaluation and Review Technique* (PERT) network analysis that the numbers u_i are *lower bounds on the expected value* of the critical path lengths of the stochastic DAG. This result dates back at least as far as Fulkerson [6], who referred to it as already being widely-known and gave a simple proof. This prompts another question: does using the actual expected values lead to superior performance in HEFT?

Unfortunately, computing the moments of the critical path length of a graph whose weights are discrete RVs was shown to be a $\#P$-complete problem by Hagstrom [7]. This means that it is generally impractical to compute the true expected values. However, efficient methods which yield better approximations than the u_i numbers are known; we discuss examples in the following two sections.

3.1.1 Monte Carlo sampling

Monte Carlo (MC) methods have a long history as a means of approximating the critical path distribution for PERT networks, dating back to at least the early 1960s [13]. The idea is to simulate the realization of all RVs (according to their pmfs) and then evaluate the critical path of the resulting deterministic graph. This is done repeatedly, giving a set of critical path instances whose empirical distribution function is guaranteed to converge to the true distribution by the Glivenko-Cantelli Theorem [2]. Furthermore, analytical results allow us to

quantify the approximation error for any given the number of realizations—and therefore the number of realizations needed to reach a desired accuracy.

The downside of Monte Carlo sampling is its cost. While modern architectures are well-suited to this approach because of their parallelism, it still may be impractical in the context of a scheduling heuristic, especially when the DAG is large; we often found this to be the case for the examples discussed in Section 5. Hence in this report we typically only use the Monte Carlo method as a means of obtaining a reference solution when estimating the critical path of G_s; see, for example, the following section.

3.1.2 Fulkerson's bound

Before introducing the alternative bounds on the critical path lengths proposed by Fulkerson, we first describe how the stochastic graph G_s can be expressed in an equivalent formulation with only edge weights. This step is not mathematically necessary but simply makes the elucidation much cleaner; it should be emphasized that all of the following still holds, with only minor adjustments, if this is not done. The most straightforward approach is to simply redefine the edge weights so that they also include the computation cost of the parent task and, if the child task is an exit, the computation cost of the child as well. More precisely, we define a new set of edge weight variables $\tilde{w}_{ik}$ which take values

$$\tilde{W}_{ik}^{ab} \coloneqq W_i^a + W_{ik}^{ab} + \delta_k W_k^b, \quad \forall a, b, i, k,$$

where $\delta_k = 1$ if t_k is an exit task and zero otherwise. Figure 5 illustrates how the graph in Figure 1 would be transformed in this manner, where the edge labels are in the form $(\tilde{W}_{ik}^{11}, \tilde{W}_{ik}^{12}, \tilde{W}_{ik}^{22}, \tilde{W}_{ik}^{21})$.

Note that $\mathbb{P}[w_{ik} = W_{ik}^{ab}] = \mathbb{P}[\tilde{w}_{ik} = \tilde{W}_{ik}^{ab}]$ for all a, b, i and k, so that $m_{ik}(\tilde{W}_{ik}^{ab}) \equiv m_{ik}(W_{ik}^{ab})$. However, we do have to make a minor adjustment to how the u_i numbers are computed since the expected value of the node weights no longer has any meaning. In particular, we set $u_i = 0$ for all exit tasks then recursively compute

$$u_i = \max_{k \in S_i} \left(u_k + \mathbb{E}[\tilde{w}_{ik}] \right) \tag{6}$$

for all others. It can readily be verified that this sequence is identical to that defined by (5) with the exception of the exit tasks, for which the corresponding numbers are now zero. (Technically we should perhaps rename this number sequence but given the fact that it is essentially identical we felt this was unnecessary.)

Now, for all $i = 1, \ldots, n$, let c_i be the critical path length from task t_i to the end and let $e_i = \mathbb{E}[c_i]$ be its expected value. Define Z_i to be the set of all weight RVs corresponding to edges downward of t_i (i.e., the remainder of the graph). Let $R(Z_i)$ be the set of all possible *realizations* of the RVs in Z_i. Given a realization $z_i \in R(Z_i)$, let $\ell(z_i)$ be the critical path length from task t_i to the end (which is

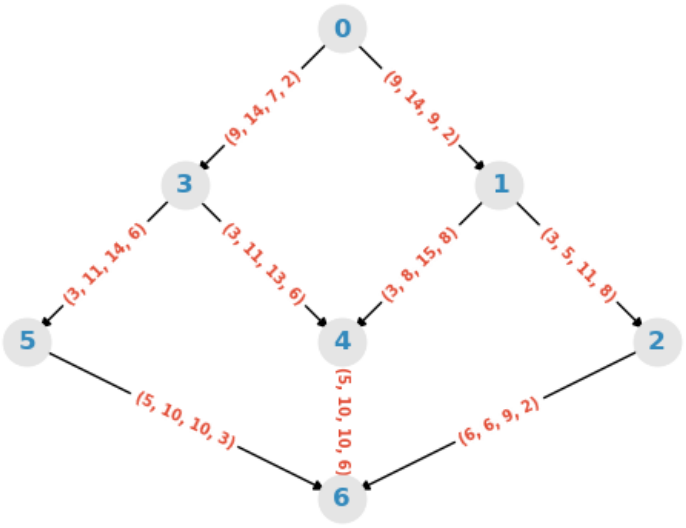


Figure 5: Edge weight-only equivalent of task DAG from Figure 1.

a scalar because all weights have been realized). Then by the definition of the expected value we have

$$e_i = \sum_{z_i \in R(Z_i)} \mathbb{P}[Z_i = z_i]\ell(z_i). \tag{7}$$

Let B_i be the set of all the weight RVs corresponding to edges which connect task t_i to its immediate children—i.e., $B_i \coloneqq \{\tilde{w}_{ik}\}_{k \in S_i}$. Further, let $R(B_i)$ be the set of all possible realizations of the RVs in B_i and let $b_i \in R(B_i)$ be any such realization. Suppose we define a sequence of numbers by $f_i = 0$, if t_i is an exit task, and

$$f_i = \sum_{b_i \in R(B_i)} \mathbb{P}[B_i = b_i] \max_{k \in S_i} \{f_k + b_{ik}\} \tag{8}$$

for all other tasks, where b_{ik} is the realization of the edge weight RV $\tilde{w}_{ik}$ under the set of realizations b_i. Then Fulkerson showed that $u_i \leq f_i \leq e_i$ holds for all i—i.e, the f_i give a tighter bound on the expected values of the critical path lengths than the u_i.

Proof of inequalities The proof proceeds by induction. Without loss of generality, we assume that the DAG has single entry and exit tasks (artificial zero-cost tasks can be added if necessary) so that in particular n is the index of the exit task. Clearly, since $u_n = f_n = e_n = 0$ by definition, the inequality holds in that case. Now for a generic task t_i we assume that the inequalities $u_k \leq f_k \leq e_k$ hold

for all $k \in S_i$ (i.e., all of t_i's child tasks) and show that the inequality $u_i \leq f_i \leq e_i$ therefore holds as well.

First, we prove the left-hand inequality. By bringing the probability into the maximization we can rewrite (8) as

$$f_i = \sum_{b_i \in R(B_i)} \max_{k \in S_i} \{\mathbb{P}[B_i = b_i](f_k + b_{ik})\}.$$

Interchanging the summation and maximization, we get the inequality

$$f_i \geq \max_{k \in S_i} \left\{ \sum_{b_i \in R(B_i)} \mathbb{P}[B_i = b_i](f_k + b_{ik}) \right\}.$$

By expanding out the term in the maximization and making use of the facts that the f_k are independent of b_i and the weight RVs corresponding to distinct edges are also independent, we see that

$$\begin{aligned}
\sum_{b_i \in R(B_i)} \mathbb{P}[B_i = b_i](f_k + b_{ik}) &= \sum_{b_i \in R(B_i)} \mathbb{P}[B_i = b_i] f_k + \sum_{b_i \in R(B_i)} \mathbb{P}[B_i = b_i] b_{ik} \\
&= f_k \sum_{b_i} \mathbb{P}[B_i = b_i] + \sum_{b_i} \mathbb{P}[B_i = b_i] b_{ik} \\
&= f_k + \sum_{b_i} \mathbb{P}[B_i = b_i] b_{ik} \\
&= f_k + \mathbb{E}[\tilde{w}_{ik}].
\end{aligned}$$

By the induction hypothesis, $f_k \geq u_k$ for all $k \in S_i$, so we have

$$\begin{aligned}
f_i &\geq \max_{k \in S_i} \{f_k + \mathbb{E}[\tilde{w}_{ik}]\} \\
&\geq \max_{k \in S_i} \{u_k + \mathbb{E}[\tilde{w}_{ik}]\} = u_i,
\end{aligned}$$

as required to establish the first part of the inequality.

Now we prove the second part. First, note that we can rewrite equation (7) as

$$e_i = \sum_{z_i \in R(Z_i)} \mathbb{P}[Z_i = z_i] \max_{k \in S_i} \{\ell(z_k) + z_{ik}\},$$

where z_{ik} is the realization of the edge weight RV $\tilde{w}_{ik}$ under the set of realizations z_i. The key now is to divide into two disjoint subsets, $Z_i = B_i \cup Y_i$, where the former is defined as before and the latter contains all of the other elements of the set (i.e., RVs corresponding to edges beyond $t_i's$ child tasks). By the independence of the edge weights, we have

$$\mathbb{P}[Z_i = z_i] = \mathbb{P}[B_i = b_i]\mathbb{P}[Y_i = y_i],$$

where y_i runs over all the possible realizations of the set Y_i. This means that

$$\begin{aligned} e_i &= \sum_{z_i \in R(Z_i)} \mathbb{P}[Z_i = z_i] \max_{k \in S_i} \{\ell(z_k) + z_{ik}\} \\ &= \sum_{b_i} \mathbb{P}[B_i = b_i] \sum_{y_i} \mathbb{P}[Y_i = y_i] \max_{k \in S_i} \{(\ell(z_k) + b_{ik})\}, \end{aligned}$$

where we change the z_{ik} to b_{ik} to acknowledge that it is now set according to the realization b_i. Interchanging the rightmost sum and its inner maximization we get the inequality

$$\begin{aligned} e_i &\geq \sum_{b_i} \mathbb{P}[B_i = b_i] \max_{k \in S_i} \left\{ \sum_{y_i} \mathbb{P}[Y_i = y_i](\ell(z_k) + b_{ik}) \right\} \\ &= \sum_{b_i} \mathbb{P}[B_i = b_i] \max_{k \in S_i} \left\{ b_{ik} + \sum_{y_i} \mathbb{P}[Y_i = y_i] \ell(z_k) \right\}, \end{aligned}$$

since the b_{ik} do not depend on y_i. By the independence assumption, for all $k \in S_i$, we have $\mathbb{P}[Y_i = y_i] = \mathbb{P}[Z_k = z_k]$. Therefore

$$\begin{aligned} e_i &\geq \sum_{b_i} \mathbb{P}[B_i = b_i] \max_{k \in S_i} \left\{ b_{ik} + \sum_{z_k} \mathbb{P}[Z_k = z_k] \ell(z_k) \right\} \\ &= \sum_{b_i} \mathbb{P}[B_i = b_i] \max_{k \in S_i} \{b_{ik} + e_k\}. \end{aligned}$$

By the induction assumption, $e_k \geq f_k$ for all $k \in S_i$, so finally we have

$$e_i \geq \sum_{b_i} \mathbb{P}[B_i = b_i] \max_{k \in S_i} \{b_{ik} + f_k\} = f_i,$$

which completes the proof.

Example As a demonstration of how the Fulkerson numbers f_i are calculated, we consider the small DAG from Figure 5. The procedure is greatly simplified in this case because the edge pmfs m_{ik} are all constant and therefore all realizations any given weight RV are equally likely. Since there are $q^2 = 4$ realizations of each edge weight, we have $|R(B_i)| = 4^{|S_i|}$ for all i. Therefore, for any task t_i and any realization $b_i \in B_i$,

$$\mathbb{P}[B_i = b_i] = \frac{1}{|R(B_i)|} = \frac{1}{4^{|S_i|}}.$$

We begin by setting $f_6 = 0$ and then working our way up the DAG. The Fulkerson numbers corresponding to tasks t_5, t_4 and t_2 are straightforward since in each case

their only child task is t_6:

$$\begin{aligned} f_5 &= \frac{1}{4}(f_6 + 5) + \frac{1}{4}(f_6 + 10) + \frac{1}{4}(f_6 + 10) + \frac{1}{4}(f_6 + 3) \\ &= \frac{5 + 10 + 10 + 3}{4} = 7, \\ f_4 &= \frac{5 + 10 + 10 + 6}{4} = 7.75, \\ f_2 &= \frac{6 + 6 + 9 + 2}{4} = 5.75. \end{aligned}$$

Things are more interesting for f_1, f_3 and f_0 since they each have two children. The calculations are quite long-winded because $|R(B_0)| = |R(B_1)| = |R(B_3)| = 4^2 = 16$, so we only include all of the details for f_1, which is computed as follows,

$$\begin{aligned} f_1 = \frac{1}{16}[&\max(f_2 + 3, f_4 + 3) + \max(f_2 + 3, f_4 + 8) + \max(f_2 + 3, f_4 + 15) + \max(f_2 + 3, f_4 + 8) \\ &+ \max(f_2 + 5, f_4 + 3) + \max(f_2 + 5, f_4 + 8) + \max(f_2 + 5, f_4 + 15) + \max(f_2 + 5, f_4 + 8) \\ &+ \max(f_2 + 11, f_4 + 3) + \max(f_2 + 11, f_4 + 8) + \max(f_2 + 11, f_4 + 15) + \max(f_2 + 11, f_4 + 8) \\ &+ \max(f_2 + 8, f_4 + 3) + \max(f_2 + 8, f_4 + 8) + \max(f_2 + 8, f_4 + 15) + \max(f_2 + 8, f_4 + 8)] \\ = \frac{1}{16}[&10.75 + 15.75 + 22.75 + 15.75 + 10.75 + 15.75 + 22.75 + 15.75 \\ &+ 16.75 + 16.75 + 22.75 + 16.75 + 13.75 + 15.75 + 22.75 + 15.75] \\ = \frac{271}{16} &\approx 16.9. \end{aligned}$$

In a similar manner, we compute $f_3 = \frac{1159}{64} \approx 18.1$ and finally $f_0 = \frac{28899}{1024} \approx 28.2$. Table 2 compares the f_i and u_i values for this example with e_i, the actual expected critical path length, which was estimated using the Monte Carlo method with 1000 realizations (see Section 3.1.1). We see that the f_i do indeed give tighter bounds than the u_i. Furthermore, if we were to rank the tasks according to the former, we would get a task priority list of $\{t_0, t_3, t_1, t_4, t_5, t_2, t_6\}$—i.e., tasks t_1 and t_3 are interchanged compared to the standard HEFT ranking. As seen in Section 2, this leads to a schedule with a smaller makespan (20 units) than the schedule obtained using the u_i numbers as ranks (22 units). Again, this is only one example, but it does support the idea that taking the f_i as task ranks rather than the u_i may be useful, a proposition that we investigate more thoroughly in Section 5.

Complexity To compute each of the f_i using (8) we need to do a lot of work, as suggested by the (deliberately) tedious breakdown of the calculations for f_1 in the example above. The problem is that the computation scales with the size of $R(B_i)$, which is exponential in the number of child tasks $|S_i|$. There are up to q^2 distinct costs for each edge so we may have $|R(B_i)| = q^{2^{|S_i|}}$. In the worst case, a task may have $n-1$ children, which can easily lead to an impractical time complexity for even relatively small values of n and q.

Table 2: Fulkerson numbers f_i for the DAG in Figure 5.

Task:	0	1	2	3	4	5	6
u_i	24.75	16.25	5.75	16	7.75	7	0
f_i	28.2	16.9	5.75	18.1	7.75	7	0
e_i	29.6	17.6	5.9	18.7	7.8	7.3	0

Fortunately, we can compute the f_i numbers more efficiently using an idea first proposed by Clingen [3] in the context of extending Fulkerson's method to the case where edge weights are modeled as continuous random variables. The key is to exploit the structure of the computations as illustrated by the example for f_1 above. In particular, we can see that certain values repeatedly arise from the maximizations—for example, in the computations above, $f_4 + 15$ is larger than the sum of f_2 and any weight realization along the edge (t_1, t_2), so it is the result of all the maximizations in which it occurs. This suggests that perhaps we can improve efficiency by calculating how frequently each term will appear in the summation, rather than working sequentially through the set of realizations $R(B_i)$. This is the intuitive basis for Clingen's method, which is as follows.

For all $i = 1, \ldots, n$ and $k \in S_i$, let $R(\tilde{w}_{ik})$ be the set of all possible realizations of the edge weight RV $\tilde{w}_{ik}$. Further, let V_i be the set of all unique values of $f_k + \tilde{w}_{ik}$ and define

$$\alpha_i = \max_{k \in S_i} (f_k + \min(R(\tilde{w}_{ik}))).$$

Let M_{ik} be the cumulative pmf along edge (t_i, t_k), so that $M_{ik}(x) = \mathbb{P}[\tilde{w}_{ik} \leq x]$, and define the related function $M^*_{ik}(x) = \mathbb{P}[\tilde{w}_{ik} < x]$. Then, if we let v run over the elements of V_i, we can rewrite equation (8) as

$$f_i = \sum_{v \geq \alpha_i} v \left(\prod_{k \in S_i} M_{ik}(v - f_k) - \prod_{k \in S_i} M^*_{ik}(v - f_k) \right). \tag{9}$$

Formally, this approach is based on the well-known fact that the cumulative probability mass function of the maximum of a finite set of (independent) RVs is equal to the product of the individual cumulative pmfs of the RVs. More intuitively, the idea is just that described above, to in some sense "count" the number of times each v is the output of a maximization: all $v < \alpha_i$ are disregarded because they can never be maximal and the modified pmf M^*_{ik} is a kind of correction term to prevent over counting.

Using Clingen's method, we can recalculate f_1 from the example above as

follows,

$$
\begin{aligned}
V_1 &= \{f_4+3, f_4+8, f_4+15, f_2+3, f_2+11, f_2+8\} \\
&= \{10.75, 15.75, 22.75, 8.75, 16.75, 13.75\}, \\
\alpha_1 &= \max\{f_4+\min(3,8,15),\ f_2+\min(3,5,8,11)\} \\
&= \max\{10.75,\ 8.75\} \\
&= 10.75, \\
f_1 &= 10.75[M_{14}(3)M_{12}(5) - M^*_{14}(3)M^*_{12}(5)] \\
&\quad + 15.75[M_{14}(8)M_{12}(10) - M^*_{14}(8)M^*_{12}(10)] \\
&\quad + 22.75[M_{14}(15)M_{12}(17) - M^*_{14}(15)M^*_{12}(17)] \\
&\quad + 16.75[M_{14}(9)M_{12}(11) - M^*_{14}(9)M^*_{12}(11)] \\
&\quad + 13.75[M_{14}(6)M_{12}(8) - M^*_{14}(6)M^*_{12}(8)] \\
&= 10.75\left[\frac{1}{4}\cdot\frac{1}{2} - 0\cdot\frac{1}{4}\right] + 15.75\left[\frac{3}{4}\cdot\frac{3}{4} - \frac{1}{4}\cdot\frac{3}{4}\right] \\
&\quad + 22.75\left[1\cdot 1 - \frac{3}{4}\cdot 1\right] + 16.75\left[\frac{3}{4}\cdot\frac{3}{4} - \frac{3}{4}\cdot\frac{1}{2}\right] \\
&\quad + 13.75\left[\frac{1}{4}\cdot\frac{1}{2} - \frac{1}{4}\cdot\frac{1}{4}\right] \\
&= \frac{271}{16} \approx 16.9.
\end{aligned}
$$

A complete description of a practical procedure for computing the Fulkerson numbers f_i using Clingen's method f_i is given in Algorithm 1.

At first blush this method may not appear to be any more efficient than before but the number of operations required to compute each of the f_i is now $O(q^2|S_i|)$, rather than the first term being exponential in the second. Of course, it should also be noted that this procedure is still more expensive than computing the u_i sequence. Indeed, as we will see in Section 5, we found that it can still be impractical when the sets of realizations for each edge are large. However, this relationship between efficiency and the number of edge weight realizations can also be exploited: for accelerated platforms which follow the model described in the previous chapter, the number of possible realizations remains small, no matter how many processors are available, so Fulkerson's numbers can still be calculated efficiently for large DAGs and target platforms with many processors.

Extensions Elmaghraby [5] proposed two refinements of Fulkerson's method. The first involves computing each of the f_i numbers in the aforementioned manner and then reversing the direction of the remaining subgraph in order to calculate an intermediate result which can be used to improve the quality of the bound. The second is a more general approach based on using two or more *point estimates* of e_i, rather than just f_i, a method that was later generalized by Robillard and

Trahan [10]. In both cases Elmagharaby proved that the new number sequences achieve tighter bounds on e_i than the Fulkerson numbers f_i. However, small-scale experimentation suggested that the improvement of Elmaghraby's new bounds over Fulkerson's were typically minor compared to the improvement of the latter over the u_i sequence so we chose to only evaluate here whether tightening the bounds at all is useful in HEFT.

Algorithm 1: Computing Fulkerson's numbers by Clingen's method.

```
for i = n, ..., 1 do
    f_i = 0, α_i = 0, V_i = {}
    for k ∈ S_i do
        r_m = ∞
        for r ∈ R(w̃_ik) do
            r_m ← min(r_m, r)
            if f_k + r ∉ V_i then
                V_i ← V_i ∪ {f_k + r}
            end
        end
        α_i ← max(α_i, f_k + r_m)
    end
    for v ∈ V_i do
        if v ≥ α_i then
            g = 1, d = 1
            for k ∈ S_i do
                g ← g × M_ik(v − f_k)
                d ← d × M*_ik(v − f_k)
            end
            f_i ← f_i + v × (g − d)
        end
    end
end
```

3.2 Adjusting the pmfs

We previously argued that in some sense the purpose of the node and edge pmfs m_i and m_{ik} is to simulate the dynamics of the processor selection phase of HEFT so that, for example, $m_i(W_i^a)$ should represent the probability that task t_i is scheduled on processor p_a, and so on. In HEFT, tasks are assigned to the processor that is estimated to complete their execution at the earliest time and attempting to model this accurately beforehand can quickly get messy and expensive—especially given the interaction between the two phases of the

algorithm. However, a sensible idea may be to simply *weight* the processor selection probabilities according to their respective computation costs: if, say, a task is 10 times faster on one processor than another then it seems more likely it will be scheduled on the former than the latter, even once the effect of contention is taken into account. With this in mind, for all tasks t_i let

$$s_i = \sum_a \frac{1}{W_i^a}$$

and define a new set of pmfs by

$$\hat{m}_i(W_i^a) = \frac{1}{W_i^a s_i} \quad \forall i, a$$

and

$$\begin{aligned} \hat{m}_{ik}(W_{ik}^{ab}) &= \hat{m}_i(W_i^a) \cdot \hat{m}_k(W_k^b) \\ &= \frac{1}{W_i^a W_k^b s_i s_k} \quad \forall i, k, a, b. \end{aligned}$$

Note that we take the reciprocal of the costs in order to reflect the idea that processors with smaller costs are more likely to be chosen than larger ones.

These modified pmfs can be used in conjunction with either upward ranking, as defined by equation (5), Fulkerson's bound, or even Monte Carlo methods. For example, in the first instance, the expectations simply become

$$\mathbb{E}[w_i] = \sum_{a=1}^{q} W_i^a \hat{m}_i(W_i^a) = \frac{q}{s_i}, \tag{10}$$

$$\mathbb{E}[w_{ik}] = \sum_{a=1}^{q} \sum_{b=1}^{q} W_{ik}^{ab} \hat{m}_{ik}(W_{ik}^{ab}) = \frac{1}{s_i s_k} \sum_{a,b} \frac{W_{ik}^{ab}}{W_i^a W_k^b}, \tag{11}$$

and these can be used with equation (5) to compute an alternative sequence of task ranks $\hat{u}_i$; of course, this is slightly more computationally expensive than computing the standard u_i ranks but only by a constant factor. Similarly, by using the modified pmfs in conjunction with equations (8) or (9) we can define alternative Fulkerson numbers $\hat{f}_i$. Table 3 presents the $\hat{u}_i$ and $\hat{f}_i$ sequences for the small example DAG from Figure 5. While the values no longer have any meaning compared to the e_i from the previous section, we see that the inequality $\hat{u}_i \leq \hat{f}_i$ holds, as we would expect (by sampling realizations according to $\hat{m}$ rather than m we could use the Monte Carlo method to estimate the corresponding number sequence $\hat{e}_i$, although that is not necessary here). The more pertinent takeaway from the table is that in both cases the value for $i = 3$ is greater than for $i = 1$, so task t_3 would be scheduled before t_1 if the sequences were used as ranks in HEFT. As seen in previous sections, this results in a smaller makespan than the

Table 3: Weighted number sequences $\hat{u}_i$ and $\hat{f}_i$ for the DAG in Figure 5.

Task:	0	1	2	3	4	5	6
$\hat{u}_i$	22.9	15.2	4.3	15.5	8.6	7.5	0
$\hat{f}_i$	25.3	15.5	4.3	17.8	8.6	7.5	0

standard HEFT algorithm. Both of these possibilities are therefore considered as alternative task ranks for HEFT in Section 5.

Note that since equations (10) and (11) are simply weighted averages we do not believe that computing $\hat{u}_i$ instead of u_i is a new idea: although we could not find any explicit references in the literature, we suspect this has been done before in practice.

4 Processor selection

Critical path estimates are used in HEFT—and many similar listing heuristics—only at the task prioritization phase. This begs the question, can they also be useful for processor selection? Arabnejad and Barbosa's Predict Earliest Finish Time (PEFT) heuristic [1] represents one sensible way this can be done. Recall from earlier chapters that before scheduling begins PEFT computes a table of *optimistic costs* C_i^a for all task and processor combinations in the following manner. First, set $C_i^a = 0$, $a = 1, \ldots, q$, for all exit tasks, then move up the DAG and recursively compute

$$C_i^a = \max_{k \in S_i} \left(\min_{b=1,\ldots,q} \left(\delta_{ab} \overline{w_{ik}} + W_k^b + C_k^b \right) \right) \tag{12}$$

for all other tasks, where $\delta_{ab} = 1$ if $a = b$ and 0 otherwise. The C_i^a values are referred to in PEFT as optimistic costs but can be interpreted as *conditional critical paths* in that they represent some estimate of future schedule costs given a processor selection. When scheduling, say, task t_i, in PEFT we choose the processor p_{opt} defined by

$$p_{opt} \coloneqq \min_a \left(F_i^a + C_i^a \right)$$

where, as in previous chapters, F_i^a is the estimated schedule makespan when t_i is completed by p_a. This is a nice extension of the dynamic programming principles underlying HEFT: rather than optimizing the schedule makespan up to the current task (i.e., F_i^a), we extend the horizon and optimize an estimate of the complete schedule makespan. Now, computing the full optimistic cost table

is only $O(n^2q)$—i.e., the same as HEFT—but since the values within are similar in nature to the upward ranks u_i it is sensible (and more efficient) to make use of them for prioritizing tasks, rather than going to the effort of computing the upward ranks as well. Hence PEFT defines task priorities C_i through

$$C_i = \frac{1}{q} \sum_a C_i^a. \tag{13}$$

It is important to note here that the task priorities as computed by (13) do not necessarily respect precedence constraints since the equality $C_i^a \leq C_k^a$ for $k \in S_i$ does not hold (because of the internal minimization over all processors). However this is easily remedied by selecting tasks for scheduling from the pool of currently "ready" tasks (i.e., those for which all of their parents have been scheduled) according to their priorities. Note also that although it is arguably more natural that task ranks include the cost of the task itself, which these do not, it is suggested that the savings made through the alternative selection step are more beneficial overall. Certainly, numerical experiments described by the original authors [1] suggest that PEFT is at least competitive with HEFT, especially when there are a large number of processing resources.

The structure of PEFT follows a more general heuristic framework that is defined by the following procedure:

1. Compute a table of conditional critical path estimates C_i^a for all $i = 1, \ldots, n$ and $a = 1, \ldots, q$.
2. Compute all task ranks C_i as some function of the C_i^a.
3. At the processor selection phase, schedule task t_i on the processor which minimizes $F_i^a + C_i^a$.

The standard heuristic is defined by using equations (12) and (13) for the first and second parts of this framework, respectively. A natural question is, are there any better choices?

4.1 Alternative conditional critical paths

All of the methods for estimating critical path lengths at the prioritization phase which were introduced previously can be modified to give conditional critical path estimates (which also disregard the cost of the task itself). No matter which, we first let $C_i^a = 0$ for $a = 1, \ldots, q$ and all exit tasks, then move up the DAG and recursively compute the other values, so from now on we focus only on the latter. The most straightforward method to extend is the optimistic lower bound, for which we now compute

$$C_i^a = \max_{k \in S_i} \left(\min_{b=1,\ldots,q} \left(C_k^b + W_k^b + W_{ik}^{ab} \right) \right), \quad a = 1, \ldots, q, \tag{14}$$

for all non-exit tasks. Note that these values are extremely similar to the optimistic costs (12) as used in PEFT, with the exception that the specific communication cost W_{ik}^{ab} is used in the minimization rather than the average $\overline{w_{ik}}$. Indeed, this is arguably the most intuitive way to define the conditional critical path since the value C_i^a is a true lower bound on the remaining makespan of any schedule which executes task t_i on processor p_a; locally-optimal processor selections are overruled if the best possible final makespan we can hope to achieve given that selection is inferior to other choices.

Alternatively, we could take a similar tack to the standard HEFT upward ranks and use an estimate of what we expect the conditional critical paths to be. More specifically, we move up the DAG and recursively compute

$$C_i^a = \max_{k \in S_i} \left(\frac{1}{q} \sum_{b=1}^{q} (C_k^b + W_k^b + W_{ik}^{ab}) \right), \quad a = 1, \ldots, q. \tag{15}$$

We can just as easily use a weighted mean inside the maximization. In particular, for all non-exit tasks we recursively compute

$$C_i^a = \max_{k \in S_i} \left(\frac{1}{\hat{s}_k} \sum_{b=1}^{q} \frac{C_k^b + W_k^b + W_{ik}^{ab}}{W_k^b + C_k^b} \right), \quad a = 1, \ldots, q, \tag{16}$$

where

$$\hat{s}_k = \sum_{b=1}^{q} \frac{1}{W_k^b + C_k^b}. \tag{17}$$

This is conceptually similar to the weighted pmf $\hat{m}$ as defined in Section 3.2, except that we also include the conditional critical path lengths in the weighting. The motivation for the change is that the probability a task t_i will ultimately be scheduled on a given processor p_a at the selection phase now also depends on the C_i^a value.

4.2 Computing task priorities

There are many different ways we can use the conditional critical path estimates C_i^a to calculate task priorities C_i. By default, PEFT uses the mean but we could use any other average over the set of processors instead. In light of our comments above, the natural equivalent of the weighted mean ranking for HEFT, as defined in Section 3.2, would be to compute the task priorities recursively, starting from the leaves, through

$$C_i = \frac{q}{\hat{s}_i} + \max_{k \in S_i} C_k, \tag{18}$$

where the maximization is taken to be zero if S_i is empty (i.e., for exit tasks) and $\hat{s}$ is as defined in (17). By adding the maximization term we ensure that the C_i give a complete valid priority list of tasks, although we could select tasks from the current set of ready tasks according to the first term as in the default PEFT algorithm instead; it makes no difference conceptually, but we use the version given here for the experiments in the following section because it is slightly more efficient in our implementation.

5 Results

In this section we use our software simulator to evaluate the alternative task prioritization phases in HEFT described in Sections 2 and 3, as well as the PEFT variants outlined in Section 4. All of the data from the experiments described here, and the code used to generate it, can be found in the Github repository associated with this chapter in the `scripts`[2] folder.

5.1 Testing environment

In order to compare two or more different heuristics, we need a suitably large and diverse set of graphs. We decided to use the same two types as in the previous chapter: a set of Cholesky graphs, with real costs based on timings from an accelerated machine that we have access to, and several sets of randomly-generated graphs based on topologies provided by the STG [11].

Since we are now interested in more general heterogeneous platforms we consider multiple sets of graphs based on randomly-generated graph topologies from the STG which differ from those in the previous chapter. In particular, a set is defined by the following parameters:

- $n \in \{100, 1000\}$, the number of non-entry/exit tasks in each of the graphs. Each graph also has a single entry and exit task so that altogether it has e.g., 102 tasks rather than 100. Once this parameter has been specified, we use the topologies of the corresponding set of that size from the STG.
- $q \in \{2, 4, 8\}$, the number of processors in the target platform.
- $\beta \in \{0.1, 1, 10\}$, the computation-to-communication ratio (CCR). Defined as before, although the manner used to generate costs that give the target CCR differs (see below).
- $h \in \{1.0, 2.0\}$, the *heterogeneity factor* of the processors. This basically determines how similar costs on different processors are to one another (again, see below for more detail).

[2] `critical-path-estimation/scripts`

- $m \in \{\mathrm{R}, \mathrm{UR}\}$, the method used to generate the costs.

Once all of the other parameters are chosen, if $m = \mathrm{UR}$ (for *unrelated*), then we use the same method as in the original HEFT paper [12] or [1] to determine all of the task computation costs on each of the processors. To wit, first an average computation cost for the entire graph $\overline{w_G}$ is chosen randomly (in our case an integer in the interval $[1, 100]$). Then, for all $i = 1, \ldots, n$, the average computation cost $\overline{w_i}$ of task t_i, is chosen uniformly at random from the interval $[0, \overline{w_G}]$. Finally, for all $a = 1, \ldots, q$, W_i^a is also chosen uniformly at random but from the interval

$$\big[\overline{w_i} \times (1 - h/2),\ \ \overline{w_i} \times (1 + h/2)\big].$$

This method is perhaps somewhat unrealistic since task costs are generated independently of whichever processor they represent; typically costs are determined at least in part by the relative processor *powers*. With this in mind, if $m = \mathrm{R}$ (for *related*), then the method proceeds by first selecting an average power $\overline{p}$ across the set of processors uniformly at random from the interval $[1, 100]$. Then for each processor p_a, $a = 1, \ldots, q$, its power r_a is in turn chosen uniformly at random from the interval

$$\big[\overline{p} \times (1 - h/2),\ \ \overline{p} \times (1 + h/2)\big].$$

Now an average task cost $\overline{t}$ is also chosen uniformly at random from the interval $[1, 100]$. For each task t_i, $i = 1, \ldots, n$, choose $x_i \in [0, 2\overline{t}]$ uniformly at random and for all $a = 1, \ldots, q$, realize $g_a \sim \Gamma(1, r_a)$ (i.e., choose g_a from a Gamma distribution with mean and variance r_a). This is done to ensure that the computation costs are not entirely determined by the power; a Gamma distribution was chosen because it is always positive and heavy-tailed, so has roughly the shape we're after. Finally, let $W_i^a = x_i g_a$ be the computation cost of task t_i on processor p_a.

Communication costs are generated in the same manner for both choices of m. First, an average edge cost $\overline{e}$ is computed such that the specified CCR is (approximately) achieved. Then for all edges (t_i, t_k), we choose $\overline{w_{ik}}$ uniformly at random from the interval $[0, 2\overline{e}]$. Then specific communication costs between the tasks on all possible pairs of distinct processors are chosen uniformly at random from the interval

$$\big[\overline{w_{ik}} \times (1 - h/2),\ \ \overline{w_{ik}} \times (1 + h/2)\big].$$

(Recall that costs are assumed to be zero when both tasks are scheduled on the same processor.) Note that the randomness here means that sometimes the target CCR is not precisely achieved, although it is usually acceptably close.

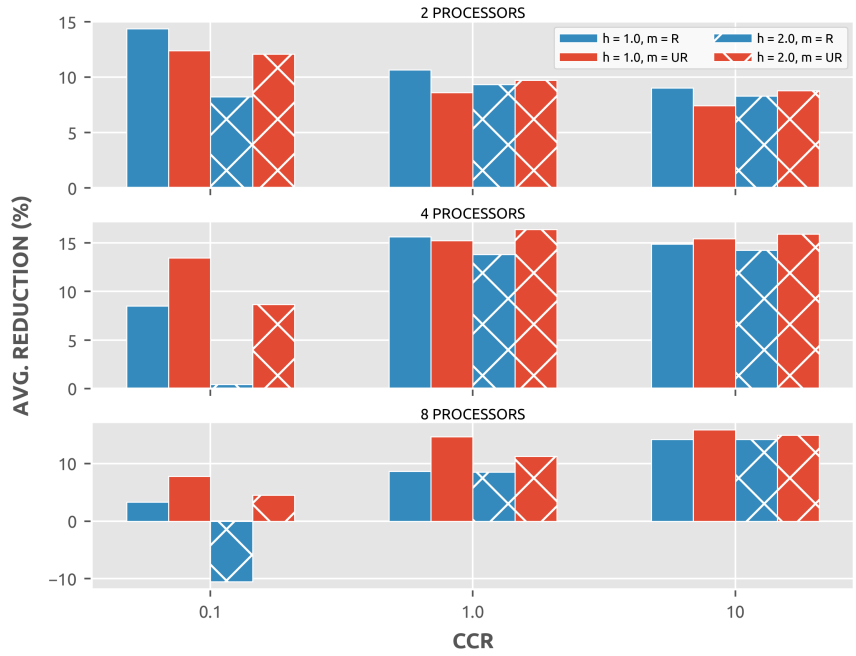


Figure 6: Average makespan reduction (%) for standard HEFT ranking phase vs random topological sort.

5.2 Benchmarking

First we want to establish how effective the HEFT task prioritization phase actually is for the graphs we consider here. More so than the speedup and schedule length ratio, which were defined and used in the previous chapter, the metric that we perhaps really want to use here is how well the task list computed by HEFT compares to other *topological sorts*—orderings which respects precedence constraints—of the tasks. Now, the `Networkx` function `topological_sort` returns a topologically sorted list of nodes in an input graph. Furthermore, the algorithm which this implements does not consider any objective other than meeting the precedence constraints, so we can in some sense regard this as a random sample from the set of all possible topological orderings for the graph. Hence to gauge the effectiveness of the HEFT task prioritization phase, we compare the makespan obtained with the standard ranking with that which would be obtained when using the task priority list returned by `topological_sort` instead. Figure 6 shows the average makespan reduction, as a percentage, for all of the subsets of DAGs from the STG with 100 tasks.

Clearly the most interesting takeaway from the figure is the negative reduction that we see for one of the DAG sets in the bottom-left corner—i.e., the random sort did better on average for those DAGs with CCR 0.1 and $h = 2.0$ for which costs were generated using the *related* method. Interestingly, the effect appears

to become more pronounced as the ratio of tasks to processors decreases: the reduction is always positive for the larger DAG sets ($n = 1000$) with the same parameters, although the raw number of instances for which the random sort outperformed standard HEFT is high for those as well.

It is not obvious why this should be so but we suspect it is related to the similar phenomenon remarked upon in the previous chapter, where HEFT failed altogether due to difficulty managing communication costs because of its greedy processor selection phase. In fact, both standard HEFT and the random ranking alternative failed altogether in the vast majority of instances for which the former was worse than the latter; the average reduction is much smaller for certain subsets because they are precisely the ones for which HEFT is most likely to struggle in general. Disregarding those instances in which both failed, the percentage of graphs for which the random sort outperformed the standard ranking was roughly 0–2% for all subsets (of both sizes). Given this, we conclude that HEFT's task prioritization phase is clearly useful except for those circumstances when the algorithm itself struggles because of its greedy selection phase.

5.3 Ranking phases in HEFT

In this section we compare the performance of HEFT with the following task prioritization phases:

- the standard upward ranks u_i,
- LB, the optimistic ranks ℓ_i as defined by (2),
- F, the Fulkerson ranks f_i as defined by (9),
- W, the weighted mean ranks $\hat{u}_i$ as defined in Section 3.2,
- WF, the weighted Fulkerson numbers $\hat{f}_i$ as defined in Section 3.2.

First, we consider the sets of DAGs based on the STG with 100 tasks as an exploratory example. Ultimately, when deciding whether to use an alternative ranking we want to know whether it's more likely to help rather than harm performance, so Figure 7 shows the difference between the percentage of DAGs for which each of the alternative rankings obtained a schedule makespan better than the standard u_i ranks and the percentage for which they were worse. Note that we combine all four of the subsets ($h = 1.0$ or 2.0, $m =$ R or UR) for each CCR and processor number combination, so that each of the bars represents a percentage over $4 \times 180 = 720$ DAGs.

Of course, the figure does not tell the whole story: what about the relative magnitudes of the makespan reductions and increases? In general, both tended to be fairly small, with makespan changes greater than 5% either way being rare.

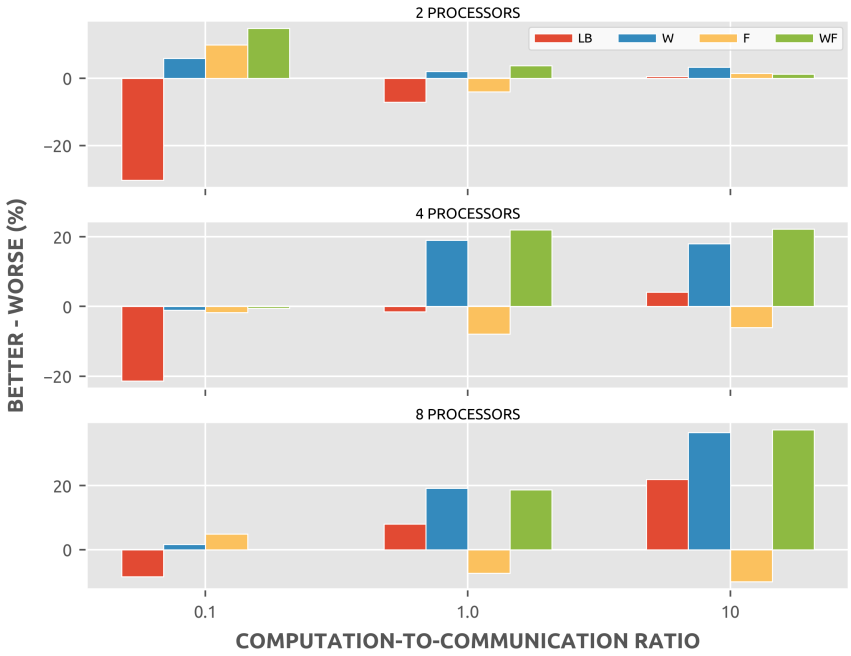


Figure 7: Percentage of instances for which the alternative rankings are better than the default, minus the percentage for which they were worse.

Overall, across the entire set, the F and WF rankings obtained an average reduction of around 1%, while the LB and F rankings on average did no better than the standard ranking. Of course, a 1% reduction might not seem significant, but in certain situations—such as when the same application has to be executed repeatedly—this may be worthwhile. Bear in mind also that this is an average value: more significant reductions can be seen, for example, as the number of processors increases. Note that while large negative average reductions can be seen for some of the rankings on certain graph sets with $\beta = 0.1$ (i.e., high communication), this is in fact down to only a handful of graphs. The issue is related to the previously-discussed problem of HEFT failing altogether in some cases. Typically, all of the rankings tend to fail for the same instances, but sometimes one or more of the rankings fails spectacularly whereas the HEFT ranking doesn't fail at all; this leads to a very large percentage difference in makespan which somewhat distorts the average. These cases are clearly important so should still be taken into account but they were very infrequent; overall, the number of failures was similar for all of the rankings.

Results so far seem to suggest that the Fulkerson ranking in particular does not seem to be an improvement on the standard ranking. Given how much more expensive it can be, this is a big problem: only large makespan reductions can really justify the extra cost. Indeed, although the weighted Fulkerson ranking WF did much better, both compared to F and the standard ranking, it was not

a significant improvement over the W ranking. This suggests that there is little advantage if any to obtaining a tighter bound on the associated stochastic graph (see Section 3.1). To confirm this, we repeated the experiments described above when the Monte Carlo approach (with a sufficient number of samples) is used to compute even tighter bounds on the critical path lengths (which are then used as ranks in HEFT). We used both the original pmfs m and the weighted pmfs $\hat{m}_i$ for this. Our results can be found in full at the Github repository but overall they supported our conclusions from the example above: tightening the bound alone led to no consistent performance gains, whereas weighting the pmfs was more promising, with a roughly 2% average makespan reduction across all of the graphs we considered.

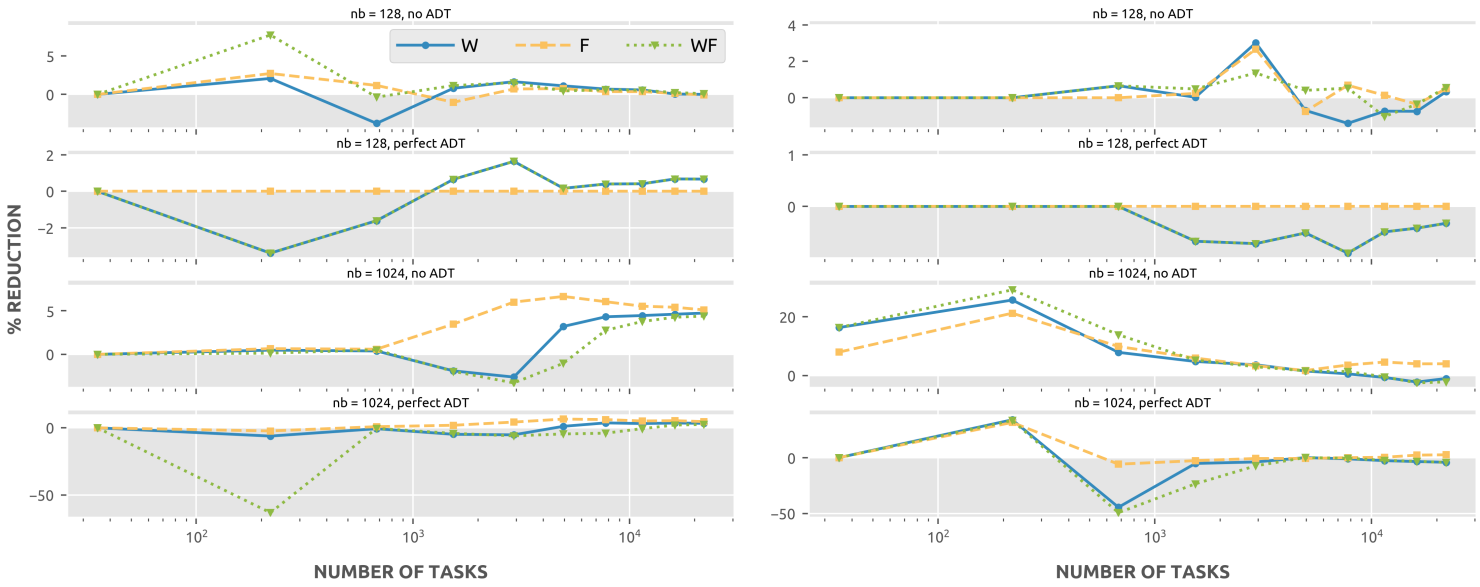
Despite this, there are situations for which the Fulkerson ranking is competitive with the standard ranking in terms of both cost and performance. In particular, for accelerated environments, such as in the previous chapter, the set of possible values each weight may take is much smaller, which significantly improves efficiency. Furthermore, the Fulkerson ranking appears to perform well for certain real application task graphs. For example, Figure 8 shows the makespan reductions of the three rankings W, F and WF compared to the default for the set of Cholesky graphs and (accelerated) target platforms defined in the previous chapter. Although there is significant variation, we see that the Fulkerson ranking is superior to the standard ranking phase overall—typically only slightly but occasionally more significantly.

While it may be worthwhile to investigate in future if the performance of the F and WF rankings improves for larger DAGs, given that the results so far suggest they offer no advantage compared to much cheaper alternatives, we decided to omit those two rankings when we repeated our comparison for sets of larger DAGs ($n = 1000$) from the STG. The metric we choose to use now is the average percentage degradation (APD), as defined in the previous chapter. Figure 9 shows the APDs for the standard HEFT ranking and the alternative LB and W rankings. We see roughly the same pattern as for the smaller DAGs: the W ranking consistently outperforms the other two, although the margins are even smaller this time than before.

5.4 Processor selection

Given the superior performance of the W ranking in the previous section, here we consider the following three variants of the PEFT framework, defined by how they compute the conditional critical paths since all use the task prioritization given by (18).

- LB, using (14).
- M, using (15).



(a) 1 GPU, 7 CPU.

(b) 4 GPU, 28 CPU.

Figure 8: Schedule makespan reduction of HEFT with alternative ranking phases for Cholesky DAGs.

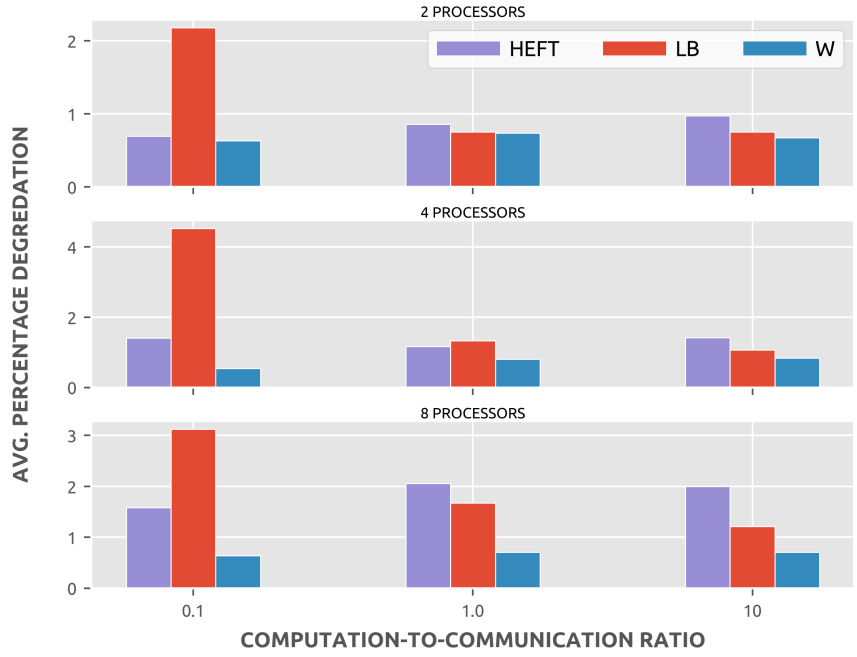


Figure 9: Average percentage degradation for select task prioritization phases in HEFT, size 1000 DAGS from the STG.

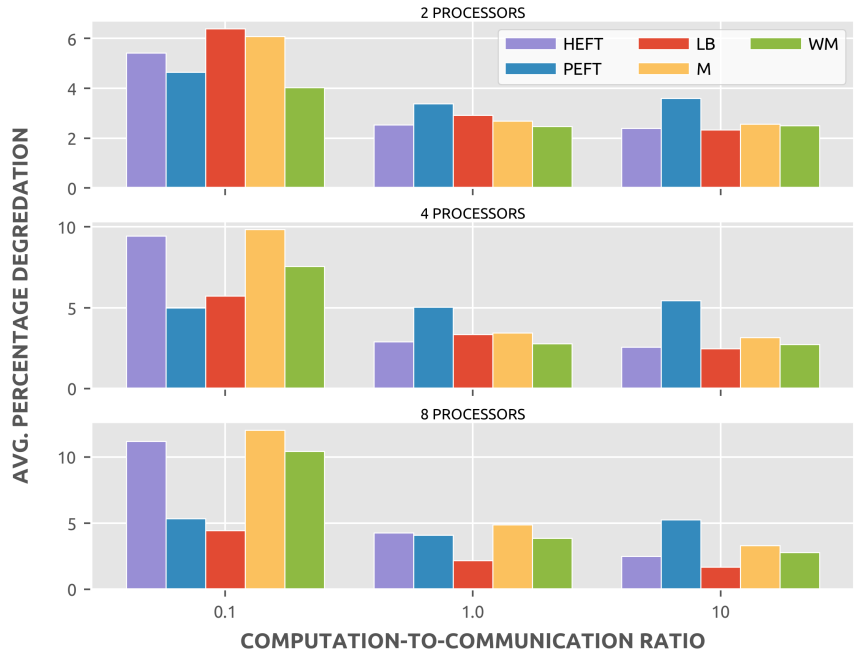


Figure 10: Average percentage degradation for PEFT variants and HEFT, size 100 DAGS from the STG.

- WM, using (16).

Figure 10 shows the average percentage degradation of these three PEFT variants, as well as the default and HEFT, for the set of DAGs based on topologies from the STG with 100 tasks (plus single entry and exit tasks). As before, we combine all four of the subsets ($h = 1.0$ or 2.0, $m =$ R or UR) so that each CCR and processor number combination represents a larger subset comprising 720 DAGs. The most immediate takeaway from the figure is that, with the exception of the subsets with the smallest CCR, standard PEFT actually does relatively poorly compared to both HEFT and the new variants. On the surface, this might seem to contradict results published elsewhere, but it should be noted that those experiments were typically for much larger and more diverse graph sets, so some local variation is to be expected. Note that when communication predominates (i.e., $\beta = 0.1$), PEFT and LB are superior to the others for both 4 and 8 processors. This is again related to the problem of listing heuristics failing altogether for DAGs with high communication: PEFT and LB recorded only around half as many failures as the others for these sets.

One obvious question we have not so far addressed is how useful the PEFT-like processor selection phase actually is compared to the simple HEFT-like earliest finish time alternative. In fact, this was far from clear in our experiments: with a few exceptions, there was usually no statistically significant difference on average between the two. The exceptions were all for the smallest CCR sets, for which

the processor selection clearly improved ($\approx 5\%$ average makespan reduction) on the ranking-only version for the LB variant and was even more significantly worse for the M and WM variants ($> 5\%$ average makespan increase).

6 Conclusions

To summarize, our main conclusions from all of the experiments described in this chapter are as follows.

- Largely supporting previous investigations along these lines [14], our biggest takeaway is perhaps that how the critical path lengths are estimated seems to make relatively little difference to the schedule makespan overall, with improvements (if any) relative to the default approach typically being fairly minor on average.
- Having said that, the W ranking was consistently more likely to result in a smaller makespan across the majority of the sets considered. The usual caveats about the limitations of our experimental framework hold of course but overall we would softly recommend it be used instead of the default, particularly since it is conceptually simple (just a weighted mean) and not significantly more expensive.
- There is no clear benefit in obtaining tighter bounds on the associated stochastic graph (i.e., Fulkerson's bound). Given the higher time complexity in the general case, this does not therefore seem to be a worthwhile alternative.
- Similarly, there appears to be little advantage in using the LB ranking, although it also wasn't clearly inferior to the standard ranking in terms of performance or cost.

References

[1] H. Arabnejad and J. G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):682–694, 2014.

[2] Louis-Claude Canon and Emmanuel Jeannot. Correlation-aware heuristics for evaluating the distribution of the longest path length of a DAG with random weights. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3158–3171, 2016.

[3] C. T. Clingen. A modification of Fulkerson's PERT algorithm. *Operations Research*, 12(4):629–632, 1964.

[4] Edward G. Coffman and Ronald L. Graham. Optimal scheduling for two-processor systems. *Acta informatica*, 1(3):200–213, 1972.

[5] Salah E. Elmaghraby. On the expected duration of PERT type networks. *Management Science*, 13(5):299–306, 1967.

[6] D. R. Fulkerson. Expected critical path lengths in PERT networks. *Operations Research*, 10(6):808–817, 1962.

[7] Jane N. Hagstrom. Computational complexity of PERT problems. *Networks*, 18(2):139–147.

[8] James E. Kelley. Critical-path planning and scheduling: Mathematical basis. *Operations Research*, 9(3):296–320, 1961.

[9] James E. Kelley and Morgan R. Walker. Critical-path planning and scheduling. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM 59 (Eastern), New York, NY, USA, 1959, pages 160–173. Association for Computing Machinery.

[10] Pierre Robillard and Michel Trahan. Technical note—expected completion time in PERT networks. *Operations Research*, 24(1):177–182, 1976.

[11] Takao Tobita and Hironori Kasahara. A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling*, 5(5):379–394.

[12] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[13] Richard M. Van Slyke. Letter to the editor—Monte Carlo methods and the PERT problem. *Operations Research*, 11(5):839–860, 1963.

[14] Henan Zhao and Rizos Sakellariou. An experimental investigation into the rank function of the Heterogeneous Earliest Finish Time scheduling algorithm. In *Euro-Par 2003 Parallel Processing*, Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, Berlin, Heidelberg, 2003, pages 189–194. Springer Berlin Heidelberg.