Improving Parallel Execution Time of Sorting on Heterogeneous Clusters

Christophe Cérin
Université de Picardie Jules Verne
LaRIA, Bat Curi, 5 rue du moulin neuf
F-80039 Amiens cedex 1- France
cerin@laria.u-picardie.fr

Michel Koskas
Université de Picardie Jules Verne
LaMFA/CNRS UMR 6140, 33 rue St Leu
F-80039 Amiens cedex 1- France
koskas@laria.u-picardie.fr

Hazem Fkaier, Mohamed Jemni
École Supérieure des Sciences et Techniques de Tunis,
Unité de recherche UTIC
5, avenue Taha Hussei, B.P. 56 Bab Menara, Tunis - Tunisie
hazem.fkaier@fst.rnu.tn, mohamed.jemni@fst.rnu.tn

Abstract

The aim of the paper is to introduce techniques in order to optimize the parallel execution time of sorting on heterogeneous platforms (processors speeds are related by a constant factor). We develop a constant time technique for mastering processor load balancing and execution time in an heterogeneous environment. We develop an analytical model for the parallel execution time, sustained by preliminary experimental results in the case of a 2-processors systems. The computation of the solution is independent of the problem size. Consequently, there is no overhead regarding the sorting problem. **Keywords:** *in-core parallel sorting algorithms, heterogeneous computing, complexity of parallel algorithms.*

1. Introduction

The advent of parallel processing, in particular in the context of *cluster computing* is of particular interest with the available technology. A special class of *non homogeneous clusters* is under concern in the paper. We mean clusters whose global performances are correlated by a multiplicative factor. We depict an heterogeneous cluster by the mean of a vector (the so call *performance vector*) set by the relative speeds of each processor.

This class of machines is of particular interest for two kinds of customers: first, for those who cannot replace instantaneously whole the components of its cluster with a new processor or disk generation but shall compose with old and new processors or disks and second for people sharing cpu-time because the cluster is not a dedicated one.

In this paper we develop techniques in order of mastering the execution time and the work of each node. The paper organization is the following. In Section 2 we summarize our previous results about sorting on heterogeneous platforms. In Section 3 we introduce the new problems. Section 4 exposes an analytical model for mastering both the parallel execution time and the work on each nodes of a 2 processors platform. Section 5 generalizes the techniques and results exposed previously. Section 6 is devoted to preliminary experimental results and Section 7 concludes the paper.

2. Our previous results

Our previous papers [2, 3, 4, 5, 1] deal with internal and external sorting algorithms on heterogeneous clusters and they are innovative since all the papers about parallel sorting algorithms (to our knowledge) always consider the special case of homogeneous computing platforms. For instance, the sorting algorithm implemented for the NAS parallel benchmark considers the homogeneous case only.

In [2, 3, 4, 5, 1] we focused on the ways to ensure good load balancing properties: if a processor is initially loaded with n integers and n is related to its performance, then the processor must never deal with more than $k.n$ data with the requirement that k should be as low as possible.

The main difference with our previous works is that we are now interested by the best way to guaranty both load balancing properties and execution time. We will explain that load balancing does not necessary implies the best execution time when we deal with heterogeneous clusters.

We focus on a specific technique, namely a meta schema for partitioning elements located on different nodes, is revisited in this paper to match an upper bound on the execution time.

2.1. Generic approaches

In our research, we specifically focus on *pivot based technique* and *one step communication* algorithms because they match the requirement of limited number of long messages of message passing programming languages in order to get performances. Thus we need a limited number of communication steps in order to avoid 'to be slowdown' by the bandwidth of the network. Let us explain the spirit of such algorithm [3] in the case of homogeneous/heterogeneous clusters. The key idea is to select pivot in the input then to partition the input according to the pivots then to exchange data then to sort.

2.2. Parallel Sorting by Over-partitioning (PSOP)

2.2.1. The homogeneous case Li and Sevcik algorithm [7] to balance the work, can potentially handle nodes that do not make uniform progress. The key ideas of Li and Sevcik [7] for the homogeneous case is to do the selection of pivots that partition the input into equal size chunks by a 'sufficient number' of random pivots that also have to partition the input into chunks of approximatively equal sizes.

The key technical discussion is about the number of pivots. The trick used in [7] is related to the following result:

Theorem 1 (See [7]) *$(p.k-1)$ pivots partition the input into $p.k$ chunks such that the size of the greatest chunk is lower or equal to n/p with probability at least*

$$1 - 2p\left(1 - \frac{1}{2p}\right)^{p.k}$$

Algorithm 1 (Code as it is found in [7])

Step 1: *initially, processor i has l_i, a portion of size n/p of the unsorted list l;*

Step 2: (selecting pivots) *a sample of $p.k.s$ candidates are randomly picked from the list where s is the oversampling ratio and k the over-partitioning ratio. Each processor picks $s.k$ candidates and passe them to a designated processor. These candidates are sorted and then $p.k-1$ pivots are selected by taking (in a 'regular way') $s^{th}, 2.s^{th}, \cdots, (pk-1)^{th}$ candidates from the sample. The selected pivots $d_1, d_2, \cdots, d_{pk-1}$ are made available to all the processors;*

Step 3: (partitioning) *since the pivots have been sorted, each processor performs binary partitioning on its local portion. Processor i decomposes l_i according to the pivots. It produces pk sublists per processor denoted l_{ij} where i, j stands for two consecutive pivots (except for the initial an final case). A sublist S_j is the union of l_{ij} with i ranging over all processors. There is pk sublists.*

Step 4: (build task queue and sort sublists) *Let $T(S_j)$ denotes the task of sorting S_j. The size of each sublist can be computed:*

$$|S_j| = \sum_{i=1}^{p} \mid l_{ij} \mid$$

Also the starting position of sublist S_j in the final sorted array can be calculated:

$$\sigma_j = 1 + \sum_{h=1}^{j-1} \mid S_h \mid$$

A task queue is built with the tasks ordered from the largest sublists size to the smallest. Each processor repeatedly takes one task $T(S_j)$ at a time from the queue. It processes the task by *(a) copying the p parts of the sublist into the final array at position σ_j to $\sigma_j + \mid S_j \mid -1$, and (b) applying a sequential sort to the elements in that range. The process continues until the task queue is empty.*

2.2.2. The heterogeneous case We have introduced in [3] and adaptation of the previous algorithm for an heterogeneous platform. The key for load balancing in this algorithm is the tasks scheduling step (step 4). The challenge is to schedule tasks with different sizes on processors having different speeds. The number of pivots required to ensure a good load balancing factor is computed according to theorem 1 but in considering that we have virtually a number of processor equal to the sum of the performance vector instead of p processors.

Concerning the scheduling step, we propose now the following strategy for implementing step 4:

- As with the original version, we compute the sizes of the tasks and then we sort them according to their sizes.
- We proceed task by task, in the decreasing order.
- We estimate the execution time of the current task on each processor as follows. Let TS_i be the size of the ith task, then the execution time on the jth processor will be: Exec_time(current task) $= (TS_i \log TS_i)/perf_j$.

  We compute, for each processor, the execution time of the tasks already scheduled on it plus the execution time of the current task: Let S_j be the sum of the execution times of the tasks already scheduled to the jth processor: $S_j = S_j +$ Exec_time(current task)
- We compare the new values of S_j relative to the different processors, and we detain the lowest value. The task will be definitely scheduled on to the processor having the lowest value of S_j and the others will ignore this task. And we pass to the next task.

We made an estimation of the execution time based on the time complexity of the sorting algorithm. But, and this point will be the main discussion point in the next section, the partitioning step considers portions of size $n/perf_i$, where $perf_i$ is the speed of processor i.

3. Discussion about the parallel execution time

We have seen that in the case of heterogeneous platforms, data are initially distributed proportionally to the speed of processors. This is the precondition of the problem.

We now examine the impact of the initial distribution or, more precisely the impact of the redistribution of data, on the parallel execution time. We determine the impact in terms of the way of restructuring the code of the meta partitioning scheme that we have introduced above. Then we justify the approach rather than processing to a redistribution of data when we start a new execution.

In previous section, when we had N data to sort on p processors depicted by their respective speeds $k_1, \cdots, k_p$, we had needed to distribute to processor p_i an amount n_i of data such that:

$$n_1/k_1 = n_2/k_2 = ... = n_p/k_p \quad (1)$$
$$\text{and } n_1 + n_2 + ... + n_p = N \quad (2)$$

The solution is:

$$\begin{array}{rcl} n_1 & = & N * k_1/(k_1 + k_2 + ... + k_p) \\ n_2 & = & N * k_2/(k_1 + k_2 + ... + k_p) \\ \ldots & = & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ n_p & = & N * k_p/(k_1 + k_2 + ... + k_p) \end{array}$$

Now, since the sequential sorts are executed on n_i data at $n_i \log n_i$ time cost (approximatively since there is a constant in front of this term), there is no reason that the nodes terminate at the same time since $n_1/k_1 \log n_1 \neq n_2/k_2 \log n_2 \neq \cdots \neq n_p/k_p \log n_p$ in this case.

The main idea that we develop now is to distribute to each processor an amount of data proportional to $n_i \log n_i$ in order to be processed by the sequential sorts. The problem is now to compute the new data sizes $n'_1, \cdots, n'_p$ such that:

$$\begin{array}{l} n'_1 + n'_2 + \cdots + n'_p = N \\ \text{and } (n'_1/k_1)\log n'_1 = (n'_2/k_2)\log n'_2 = \cdots = \\ \hfill (n'_p/k_p)\log n'_p \end{array}$$

First of all, we show that this new distribution converges to the initial distribution when N tend to infinity. The fundamental reason is that $\lim\limits_{n\to\infty} \dfrac{n}{\log n} = \lim\limits_{n\to\infty} (n)$.

Let us simply consider the case of 2 processors at speed k_1, k_2 to get the intuition that the distributions converge. The first (when we consider the initial method) distribution is:

$$\begin{aligned} n_1 &= N * k1/(k1+k2) \\ n_2 &= N * k2/(k1+k2) \end{aligned}$$

and consequently $(k1/k2) = (n_1/n_2)$. The new distribution is:

$$\begin{aligned} n'_1 + n'_2 &= N \\ (n'_1/k_1)\log n'_1 &= (n'_2/k_2)\log n'_2 \end{aligned}$$

and consequently

$$\frac{k_1/k_2}{=} \frac{n'_1 \log n'_1}{(n'_2 \log n'_2} = \frac{n'_1}{n'_2} * \frac{\log n'_1}{\log n'_2}$$

If N increases, then n_1 and n_2 will also increase. And more they increase, more the ratio of log terms tends to 1. Consequently, the ratio n'_1/n'_2 tends to k_1/k_2 which is equal to n_1/n_2. Consequently, more N is high, more the execution times of sorts will be close each together.

Second, let us consider some numerical examples to see if some practical cases could be solved efficiently by our new method despite the fact that asymptotically speaking it is equivalent to the previous one.

Let P_1, P_2 two processors characterized by their speeds $k_1 = 1, k_2 = 6$ respectively and let $N = 321$. The first method gives $n_1 = 46$ and $n_2 = 275$ whereas the second method gives (approximatively):

$$\begin{aligned} &n'_1 = 64, (64\log 64)/1 = 64*6 = 384 \\ &n'_2 = 257, (257\log 257)/6 = \tfrac{257*9}{6} = 385,5 \end{aligned}$$

The ratio $k_1/k_2 = 0,1666$ whereas the ratio $n'_1/n'_2 = 0,249$. The unbalance is $1 - 0.1666/0.249 = 37\%$... which is important. Let us consider the case of Giga bytes of data and let $N = 419G$. The first distribution gives $n_1 = 60G$ and $n_2 = 359G$. The second distributions gives:

$$\begin{aligned} n'_1 = 64G, \quad & (64G\log 64G)/1 = 64G*36 \\ & = 2.473.901.162.496 \\ n'_2 = 355G, \quad & \tfrac{355G\log 355G}{6} = \tfrac{355G*39}{6} \\ & = 2.477.659.258.880 \end{aligned}$$

The n'_1/n'_2 ratio is 0180, thus the unbalance factor is now 8%.

We are now convinced that when N tends to infinity, the n'_1/n'_2 ratio tends to $1/6$, that is to say to n_1/n_2. Moreover, we think that we have many situations that could benefit from our optimization.

4. Computation of the optimal sizes (2 processor case)

We are going to compute the n'_i sizes by approximating the solution through a Taylor development for the log function. Note that with our initial method we have $n_i = N\dfrac{k_i}{(k_1 + \cdots + k_m)}$.

Let us find n'_i under the form $n'_i = N\dfrac{k_i}{k_1 + \ldots + k_m} + a_i\dfrac{N}{\log N}$.

The idea is to produce a new term of form $a_i\frac{N}{\log N}$ complementing the previous distribution.

In fact, we will now compute the a_i terms uniquely for the case of two processors. Moreover, it is clear that the sum of a_i terms is null since the sum of n_i terms is invariant and is equal to N.

Theorem 2 *The approximated sizes to distribute to each processor of a two processors cluster of speed $k_1, k_2 \in \mathbb{N}$ and for a problem size of n data are:*

$$n_1 = n\frac{k_1}{k_1+k_2} + \frac{n}{\log n}\log\left(\frac{k_2}{k_1}\right)\frac{k_1k_2}{(k_1+k_2)^2}$$

and

$$n_2 = n\frac{k_2}{k_1+k_2} + \frac{n}{\log n}\log\left(\frac{k_1}{k_2}\right)\frac{k_1k_2}{(k_1+k_2)^2}.$$

Proof: see appendix.

5. General case

In this section we compute the optimal sizes in the general case of any number of processors. We reuse the previous assumptions and techniques. Let us assume that K is the sum of the relative speeds k_i of a p processor system. The problem is depicted by the following 3 equations:

$$n_i = \frac{k_i}{K}N + \epsilon_i, \quad (1 \le i \le p) \tag{3}$$

$$\sum_{i=1}^{p} \epsilon_i = 0 \tag{4}$$

$$\frac{n_i \log n_i}{k_i} = \frac{n_j \log n_j}{k_j}, \quad (1 \leq i,j \leq p) \qquad (5)$$

Note again that Equation 4 says that the sum of the correcting factors should be null because the sum of the n_i is invariant and is equal to N. We develop Equation 5 and we reuse the same facts during the approximation that we have done in the previous section. In another words, we proceed again by equivalence.

Theorem 3 *The approximated sizes to distribute to each processor of a cluster of processor speed* $k_i \in \mathbb{N}$ *and for a problem size of* n *data are:*

$$n_i = n\frac{k_i}{K} + \frac{n}{\log n}\left[\frac{k_i}{K^2}\sum_{j=1}^{p} k_j \log\left(\frac{k_j}{k_i}\right)\right]$$

Proof: see appendix.

6. Experimental results (preliminary)

In order to get trends from our new partitioning schema, we have modified one of our code [6] and we have experimented on a 2 processor system (2 Alpha 21164 EV 56 processors[1] at 533Mhz interconnected with Fast Ethernet under MPI). The code is an implementation of Parallel Sample Sort (PSS) for heterogeneous platforms.

We set arbitrary the performance vector to $\{1,3\}$ (processor 1 is 3 times faster than processor 0) and we measure the sizes of data on each node during the last step (the sequential sorting step) for our new approach and for the previous one. We execute our codes for an input size of 1048576 integers with an oversampling ratio of 6 (see [6] for an explanation of the role of this coefficient).

According to our new schema, we measure a deviation of $+6.7\%$ and -2.6% comparing to the optimal sizes (given as $262144 + 15580 = 277725$ integers and 770851 integers) and for processor 0 and 1 reciprocally. These results corresponds to means.

According to our previous schema, we measure a deviation of -13.4% and -1.5% comparing to the optimal sizes (given as 277725 integers and 770851 integers) and for processor 0 and 1 reciprocally. These results corresponds to means.

Thus, we find that our new schema does not improve significantly the load balance factor for PSS. It is partly expected because we know that PSS has, by nature, important deviation regarding the load balancing factor. Here the load balancing factor has a deviation of about 14% (in mean); we have measured 15% of deviation on a 4 processors system [6] and according to a performance vector set to $\{8,5,3,1\}$.

In fact, such deviation is due to the number of selected pivots and the way they are selected. We conclude that our new schema can not compensate totally the deviation of PSS. However, we notice that if we consider individually the results on each node, our schema provides less dispersion than for our previous schema. It is promising.

We are currently experimenting with Parallel Sort by Over-partitioning which has good properties in terms of load balancing and execution time [6]. We are now convinced that improvements in performance are possible for such technique. In fact, this project is part of the Grid-Explorer project[2] aiming of building a platform of 1000 CPU in order to emulate grid systems. The platform will be available on November 2004. The heterogeneity of CPU could be obtained by the use of FreqCPU Linux module[3]. Cpufrefd is a small daemon to adjust cpu speed and voltage for kernels using any of the cpufreq drivers available. The CpuFreq patch has been included in the 2.6 kernel tree. This module will allows us to configure an heterogeneous platform from an homogeneous one. So we will check how much time we would obtain in our comparisons if we use the previous and the new technique in a real system.

7. Conclusion

In this paper we have demonstrated how to optimize the data distribution in the case of sorting on heterogeneous platforms. Previous works on this subject have concentrated their efforts on load balancing but not on execution time criteria. This paper makes the bridge between the two required properties. The new technique is a constant

1 The system is homogeneous but we measure the data size on each node and not the execution time

2 See: `http://www.lri.fr/ fci/GdX`

3 See: `http://sourceforge.net/projects/cpufreqd`

time one. Consequently it introduces no overhead regarding the sorting problem.

We would like also to mention that the technique used in this paper forms a meta-partitioning schema, generalizing our previous works, and it can be reused in the case where the sequential brick of the parallel algorithm has a time complexity different than $n \log n$. Imagine for instance that we have a problem that can be divided into independent portions and that we have to use a sequential algorithm of time complexity $\sqrt{n}$.

We would like to execute the parallel algorithm on an heterogeneous cluster of p processors. The problem here is to find the $n_i, (1 \leq i \leq p)$ such that $n = n_1 + \cdots + n_p$ and $\sqrt{n_1}/k_1 = \cdots = \sqrt{n_p}/k_p$. We are guessing that Taylor development of $\sqrt{n+1}$, i.e. $(1 + 1/2 * n - 1/8 * n^2 + 1/16 * n^3 - 5/128 * n^4 + 7/256 * n^5 + \mathcal{O}(n^6))$ could be used in this case.

So, we are planning to solve the problem of computation of the n_i values for a large scale of usual time complexity. In this way, one can imagine that our meta-partitioning schema will organize itself according to a sequential portion of code and for the purpose of distributing data in an efficient way.

In the future, we will also consider the problem of finding a solution for the partitioning problem when the complexity of the sequential algorithm cannot be expressed with rational fractions. Dynamic programming could certainly offer solutions with an overhead that should be maintain as low as possible. We think that the main interest in using dynamic programming for our problem resides in the fact that we may assume that the cost function, here $n \log n$, is not necessary decomposable under rational fractions.

References

[1] C. Cérin. An out-of-core sorting algorithm for clusters with processors at different speed. In *16th International Parallel and Distributed Processing Symposium (IPDPS), Ft Lauderdale, Florida, USA*, page Available on CDROM from IEEE Computer Society, 2002.

[2] C. Cérin and J.-L. Gaudiot. Evaluation of two bsp libraries through parallel sorting on clusters. In *Proceedings of WCBC'00 (The Second International Workshop on Cluster-Based Computing) in conjuction with ICS'00 (International Conference on Supercomputing, sponsored by ACM/SIGARCH)*, pages pp 21–26, Santa Fe, New Mexico, 6 May 2000.

[3] C. Cérin and J.-L. Gaudiot. An over-partitioning scheme for parallel sorting on clusters running at different speeds. In *Cluster 2000. IEEE International Conference on Cluster Computing. Technische Universität Chemnitz, Saxony, Germany. (Poster)*, 28 Nov. - 2 Dec. 2000.

[4] C. Cérin and J.-L. Gaudiot. Parallel sorting algorithms with sampling techniques on clusters with processors running at different speeds. In *HiPC'2000. 7th International Conference on High Performance Computing. Bangalore, India*, Lecture Notes in Computer Science. Springer-Verlag, 17-20 Dec. 2000.

[5] C. Cérin and J.-L. Gaudiot. On a scheme for parallel sorting on heterogeneous clusters. *FGCS (Future Generation Computer Systems*, 18(issue 4), 2002. The special issue is preliminary scheduled for publication in future vol.

[6] C. Cérin, M. Jemni, and H. Fkaier. A synthesis of parallel out-of-core sorting programs on heterogeneous clusters. In *3st International Symposium on Cluster Computing and the Grid, Tokyo*, May 2003.

[7] H. Li and K. C. Sevcik. Parallel sorting by overpartitioning. In *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, pages 46–56, New York, NY, USA, June 1994. ACM Press.

Appendix

Proof of Theorem 2: let

- $\alpha_1 = k_1/(k_1 + k_2)$
- $\alpha_2 = k_2/(k_1 + k_2)$
- $\alpha = k_1 k_2/(k_1 + k_2)$
- a_1 and a_2 be the two unknown terms (and $a_1 = -a_2$)

Let us recall the following Taylor developments:

- $\text{Taylor}(\log(1-x)) = -x - 1/2 * x^2 - 1/3 * x^3 - 1/4 * x^4 - 1/5 * x^5 - 1/6 * x^6 + \mathcal{O}(x^7)$
- $\text{Taylor}(\log(1+x)) = x - 1/2 * x^2 + 1/3 * x^3 - 1/4 * x^4 + 1/5 * x^5 - 1/6 * x^6 + O(x^7)$

We proceed by equivalence starting from the fact expressing that the execution times are identical:

$$\begin{aligned}
&(n_1 \log n_1)/k_1 = (n_2 \log n_2)/k_2 \iff \\
&((\alpha_1 n + a_1)\log(\alpha_1 n + a_1))/k_1 - ((\alpha_2 n + a_2)\log(\alpha_2 n + a_2))/k_2 = 0 \iff \\
&k_2(\alpha_1 n + a_1)\log[\alpha_1 n(1 + a_1/(\alpha_1 n)] - k_1(\alpha_2 n - a_1)\log[\alpha_2 n(1 - a_1/(\alpha_2 n)] = 0 \iff \\
&k_2\alpha_1 n \log(\alpha_1 n) + k_2 a_1 \log(\alpha_1 n) + k_2\alpha_1 n \log(1 + a_1/(\alpha_1 n)) + \\
&k_2 a_1 \log(1 + a_1/\alpha_1 n) - \\
&\text{(this term is close to 0 when we take the Taylor development, so we neglish it)} \\
&k_1\alpha_2 n \log(\alpha_2 n) + k_1 a_1 \log(\alpha_2 n) - k_1\alpha_2 n \log(1 - a_1/(\alpha_2 n)) + k_1 a_1 \log(1 - a_1/(\alpha_2 n)) \\
&\text{(this term is close to 0 when we take the Taylor development, so we neglish it)} \iff \\
&\alpha n \log(\alpha_1 n/(\alpha_2 n)) + \alpha \log((1 + a_1/(\alpha_1 n))/(1 - a_1/(\alpha_2 n))) + \\
&k_2 a_1 \log(\alpha_1 n) + k_1 a_1 \log(\alpha_2 n) = 0 \iff \\
&\text{(We use Taylor development here for logs)} \\
&\alpha n \log(k_1/k_2) + \alpha n(a_1/(\alpha_1 n) + a_1/(\alpha_2 n)) + k_2 a_1 \log(\alpha_1 n) + k_1 a_1 \log(\alpha_2 n) = 0 \iff \\
&\alpha n \log(k_1/k_2) + \alpha n a_1/n((k_1 + k_2)^2/(k_1.k_2)) + a_1(k_1 \log(\alpha_1 n) + k_1 \log(\alpha_2 n)) \iff \\
&\alpha n \log(k_1/k_2) + a_1[(k_1 + k_2) + k_2 \log(\alpha_1 n) + k_1 \log(\alpha_2 n)] = 0 \iff \\
&a_1 = \frac{k_1 k_2/(k_1 + k_2) n \log(k_2/k_1)}{(k_1 + k_2) + k_2 \log(\alpha_1 n) + k_1 \log(\alpha_2 n)}
\end{aligned} \tag{6}$$

To observe the result, we have now to show that the denominator is close to $(k_1 + k_2)\log(n)$ which is not very difficult to obtain. In this case, we obtain the aforementioned result:

$$a_1 = \frac{k_1 k_2}{(k_1 + k_2)^2} \log(k_2/k_1) \frac{n}{\log n} \tag{7}$$

A similar computation leads to the value of n_2. To summarize, we propose to use the following values for n_1, n_2:

$$n_1 = n\frac{k_1}{k_1 + k_2} + \frac{n}{\log n} \log\left(\frac{k_2}{k_1}\right) \frac{k_1 k_2}{(k_1 + k_2)^2}$$

and

$$n_2 = n\frac{k_2}{k_1 + k_2} + \frac{n}{\log n} \log\left(\frac{k_1}{k_2}\right) \frac{k_1 k_2}{(k_1 + k_2)^2}.$$

Proof of Theorem 3: we proceed again by equivalence from the previous equations.

$$\begin{aligned}
&k_j\left(\frac{k_i}{K}N + \epsilon_i\right)\log\left(\frac{k_i}{K}N + \epsilon_i\right) = k_i\left(\frac{k_j}{K}N + \epsilon_j\right)\log\left(\frac{k_j}{K}N + \epsilon_j\right) \iff \\
&\frac{k_i k_j}{K}N\left[\log\left(\frac{k_i}{K}N + \epsilon_i\right) - \log\left(\frac{k_j N}{K} + \epsilon_j\right)\right] + k_j\epsilon_i \log\left(\frac{k_i}{K}N + \epsilon_i\right) - k_i\epsilon_j \log\left(\frac{k_j}{K}N + \epsilon_j\right) = 0 \iff \\
&\frac{k_i k_j}{K}N\left[\log\left(\frac{k_i}{k_j}\right) + \frac{\epsilon_i K}{k_i N} - \frac{\epsilon_j K}{k_j N}\right] + k_j\epsilon_i \log\left(\frac{k_i}{K}N\right) - k_i\epsilon_j \log\left(\frac{k_j}{K}N\right) = 0 \iff \\
&\frac{k_i k_j}{K}N \log\left(\frac{k_i}{k_j}\right) + \epsilon_i\left[k_j + k_j \log\left(\frac{k_i}{K}N\right)\right] - \epsilon_j\left[k_i + k_i \log\left(\frac{k_j}{K}N\right)\right] = 0 \iff \\
&\epsilon_j = \frac{\frac{k_i k_j}{K}N \log\left(\frac{k_i}{k_j}\right) + \epsilon_i\left(k_j + k_j \log\left(\frac{k_i}{K}N\right)\right)}{k_i\left(1 + \log\left(\frac{k_j}{K}N\right)\right)}
\end{aligned} \tag{8}$$

Now, we have also that:

$$\epsilon_1 + \sum_{j=2}^{p} \epsilon_j = 0$$

Thus,

$$\begin{aligned}
&\epsilon_1 \left[\sum_{j=2}^{p} \frac{k_j \left(1+\log\left(\frac{k_1}{K}N\right)\right)}{k_1 \left(1+\log\left(\frac{k_j}{K}N\right)\right)} \right] + \frac{k_1}{Kk_1} N \sum_{j=2}^{p} \frac{k_j \log\left(\frac{k_1}{k_j}\right)}{1+\log\left(\frac{k_j}{K}N\right)} = 0 \iff \\
&\epsilon_1 \left[1 + \frac{1+\log\left(\frac{k_1}{K}N\right)}{k_1} \sum_{j=2}^{p} \frac{k_j}{1+\log\left(\frac{k_j}{K}N\right)} \right] + \frac{N}{K} \sum_{j=2}^{p} \frac{k_j \log\left(\frac{k_1}{k_j}\right)}{1+\log\left(\frac{k_j}{K}N\right)} = 0 \iff \qquad (9) \\
&\epsilon_1 \sum_{j=1}^{p} \frac{k_j}{1+\log\left(\frac{k_j}{K}N\right)} + \frac{k_1 N}{K\left(1+\log(\frac{k_1}{K}N)\right)} \sum_{j=1}^{p} \frac{k_j \log\left(\frac{k_1}{k_j}\right)}{1+\log\left(\frac{k_j}{K}N\right)} = 0
\end{aligned}$$

Now we use the facts that $1+\log(k_1/KN) \sim \log N$, $1+\log(k_j/KN) \sim \log N$ and

$$\sum_{j=1}^{p} \frac{k_j}{1+\log\left(\frac{k_j}{K}N\right)} \sim K/\log N$$

to derive that

$$\epsilon_1 = \frac{k_1 N}{K^2 \log N} \sum_{j=1}^{p} k_j \log\left(\frac{k_j}{k_1}\right)$$

Hence the formula:

$$\epsilon_i = \frac{N}{\log N} \left[\frac{k_i}{K^2} \sum_{j=1}^{p} k_j \log\left(\frac{k_j}{k_i}\right) \right] \qquad (10)$$

Verification: for the 2 processors case we get the previous result $\epsilon_1 = \frac{N}{\log N} \log\left(\frac{k_2}{k_1}\right) \frac{k_1 k_2}{(k_1+k_2)^2}$ as expected.