# Efficient Parallelized Network Coding for P2P File Sharing Applications

Karam Park[1], Joon-Sang Park[2], Won W. Ro[1]

[1] School of Electrical and Electronic Engineering
Yonsei University, Seoul, Korea
{riopark, wro}@yonsei.ac.kr

[2] Department of Computer Engineering
Hongik University, Seoul, Korea
jsp@hongik.ac.kr

**Abstract.** In this paper, we investigate parallel implementation techniques for network coding to enhance the performance of Peer-to-Peer (P2P) file sharing applications. It is known that network coding mitigates peer/piece selection problems in P2P file sharing systems; however, due to the decoding complexity of network coding, there have been concerns about adoption of network coding in P2P file sharing systems and to improve the decoding speed the exploitation of parallelism has been proposed previously. In this paper, we argue that naive parallelization strategies of network coding may result in unbalanced workload distribution and thus limiting performance improvements. We further argue that higher performance enhancement can be achieved through load balancing in parallelized network coding and propose new parallelization techniques for network coding. Our experiments show that, on a quad-core processor system, proposed algorithms exhibit up to 30% of speed-up compared to an existing approach using 1 Mbytes data with 2048×2048 coefficient matrix size.

**Keywords:** Network coding, parallelization, random linear coding.

## 1 Introduction

Multi-core systems nowadays are prevalent; they are found in a wide spectrum of systems, from high performance servers to special purpose embedded systems. Recently, the trend has been embedding more and more cores in a processor rather than increasing clock frequency rate to boost processors' performance [1]. In this paper, we propose new implementation techniques that can enhance the performance of network coding [2] by fully exploiting parallelism on the multi-core systems.

Network coding which is generally due to Ahlswede *et al.* [2] is a method that can be used to enhance network throughput and reliability. In addition, it has been shown that network coding benefits peer-to-peer (P2P) file sharing [3], especially so-called file swarming type systems. In file swarming systems, a file is divided into multiple pieces and pieces are exchanged among peers. To download a file, a peer must collect all the pieces comprising the file. If a peer downloads multiple pieces simultaneously

from peers, it dramatically reduces downloading delay, which is the main advantage of using the file swarming technique. However, the selection of peers and pieces to download has a big impact on the overall performance, which is generally referred to as the *piece selection problem*. The use of network coding mitigates this problem in P2P file swarming systems [3]. In network coded systems, the data are "encoded" into packets such that the packets are equally important, i.e., no difference exists among the packets being exchanged, and thus a peer is only suppose to collect a specific number of equally important packets

One pitfall of network coding is computational overhead. Original data are coded before exchanging and downloaded packets are to be decoded to recover the original information. The decoding process is implemented usually as a variation of Gaussian elimination which has $O(n^3)$ computational complexity. This complexity is quite pricy in fact especially when the size of the file is huge. It is probable that the time spent for decoding may actually cancel out all the benefit of reduced transmission time. Thus, it is critical for network coded P2P systems to have a fast enough decoder. To provide fast decoding speed, Shojania *et al.* has suggested *Parallelized Progressive Network Coding (PPNC)* [4]. However, due to its unbalanced workload on each parallel task (or thread), their algorithms cannot take full advantage of parallelism.

In this paper, we propose parallel implementations of network coding that in nature balance workload among parallel tasks. Via real machine experiments, we show that our new techniques allow meaningful reduction of execution time compared to *PPNC*. On a quad core system for example, we achieve speed-up of 3.25 compared with a serial implementation and 30% of performance improvement over *PPNC* algorithm with 1Mbytes data and the coefficient matrix size of 2048×2048.

## 2 Background

In this section, we present an introduction of network coding and related work.

### 2.1 Principles of Network Coding

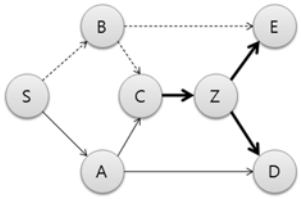


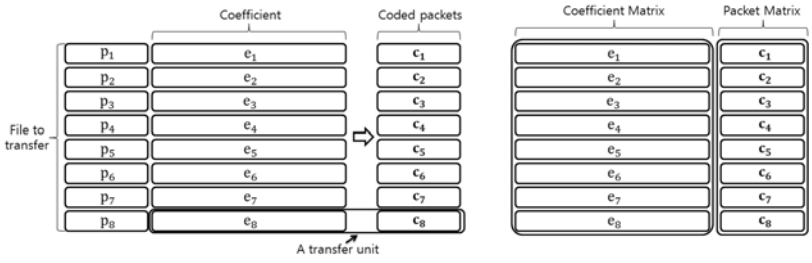

**Fig. 1.** A Communication Networks for Network Coding

Fig. 1 depicts a directed graph representing a simple communication network; the edges represent pathways for information transfer and the node *S* is the source, and

the node *D* and *E* represent receivers. The other remaining nodes represent intermediate points in the routing paths.

In this example, network coding enables us to multicast two bits per unit time assuming that each link conveys a bit per unit time, which cannot be achieved without network coding, i.e., through traditional routing. Suppose we generate data bits *a* and *b* at source *S* and want to send the data to both *D* and *E*. We send data *a* through path *SAC*, *SAD*, and data *b* through *SBC*, *SBE*. With the routing, we can only send either *a* or *b* but not both, from *C* to *Z*. Suppose we send data *a* to *Z*. Then *D* would receive *a* twice from *A* and *Z*, and would not get *b*. Sending *b* instead would also raise the same problem for *E*. Therefore, routing is insufficient as it cannot send both data *a* and *b* to both *D* and *E* simultaneously. Using network coding, on the other hand, we could encode the data *a* and *b* received in *C* and send the encoded version to *CZ*. Say we use bitwise xor for encoding. Then, *a* and *b* are encoded to '*a* xor *b*'. The encoded data is sent along on the path *CZD* and *CZE*. Node *D* receives data *a* and '*a* xor *b*', so it can decode *b* from them. It is the same for node *E*, where it receives data *b* and '*a* xor *b*'.

However, to assume the increased throughput that network coding allows, the encoding/decoding process must not be the bottleneck. The encoding/decoding process depends on the coding solution to be used and there are several ways to find out an optimal coding solution given a network. In this paper we restrict ourselves to the random linear coding [6][7], since it is the most widely used coding solution which is asymptotically optimal in any network. Now we explain how encoding and decoding works in random linear coding.




(a) (b)

**Fig. 2.** Encoding Concept and Received Data Structure

Let us assume that an application transfers a file. Then the file is divided into a specific *number of blocks* as shown in Fig. 2-(a) where $\mathbf{p}_k$ denotes $k^{th}$ block. A coded packet $\mathbf{c}_i$ is a linear combination of the blocks constituting the file. That is $\mathbf{c}_i = \sum_{k=1}^{n} \mathbf{e}_k \mathbf{p}_k$, where $n$ is the number of blocks and the coefficient $\mathbf{e}_k$ is a certain element randomly chosen in a certain finite field **F**. Every arithmetic operation is over the field **F**. The coded packet $\mathbf{c}_i$ is broadcasted to other destination nodes along with the coefficient vector, $[\mathbf{e}_1, \ldots, \mathbf{e}_n]$, stored in the header. This "transfer unit" is shown in Fig. 2-(a).

On reception of coded packets, nodes in the path to the destinations re-encode the coded packets and send them to downstream nodes. When a coded packet reaches a destination node it has to be stored in the local memory. For the destination node to decode the packets and recover the original file, it needs to get *n* transfer units with independent coefficient vectors. Let say a receiver has collected *n* transfer units and let $\mathbf{E}^{\mathrm{T}} = [\,\mathbf{e}_1^{\mathrm{T}} \ldots \mathbf{e}_n^{\mathrm{T}}\,]$, $\mathbf{C}^{\mathrm{T}} = [\,\mathbf{c}_1^{\mathrm{T}} \ldots \mathbf{c}_n^{\mathrm{T}}\,]$ and $\mathbf{P}^{\mathrm{T}} = [\,\mathbf{p}_1^{\mathrm{T}} \ldots \mathbf{p}_n^{\mathrm{T}}\,]$ where superscript T stands for the transpose operation. As the coded packet was calculated as $\mathbf{C} = \mathbf{EP}$, we can recover the original file $\mathbf{P}$ from $\mathbf{C}$ by $\mathbf{P} = \mathbf{E}^{-1}\mathbf{C}$. Note that $\mathbf{E}$ needs to be invertible, so all coefficient vectors $\mathbf{e}_k$'s must be independent with each other. Usually a variant of Gaussian elimination is used to recover $\mathbf{P}$. When transfer units arrive to a destination, it organizes coefficient and packet matrixes as Fig. 2-(b) as a preparation for running Gaussian elimination. A typical Gaussian elimination or LU decomposition restricts us to wait until we collect *n* transfer units and have the *n*×*n* coefficient matrix before start running the process. However, with progressive decoding [6], we have no need to wait until all transfer units received. Rather decoding is done progressively as each transfer unit is arrived.

Since the decoding takes $O((n + m) \times n^2)$ time where *m* is the block size, *m* and *n* are important parameters and given the file size l, *n* and *m* are inverse proportional to each other since l = *n* * *m*. In the file swarming scenarios, the bigger *n* enables the greater downloading delay reduction, since a peer can receive at most *n* simultaneous block transfers reducing the downloading delay by *n*. But since the decoding delay which might cancel out the downloading delay benefit increases proportional to $n^3$, fast decoding implementation is a key to get the benefit comes with a large *n*. In other words, given a fast decoding algorithm, a larger *n* allows a bigger performance gain.

### 2.2 Related Works

Ahlswede *et al.* first introduced the network coding and showed the usefulness of network coding [2]. Koetter and Medard proved later that in a network, the maximum throughput can be achieved with linear network codes [5]. With these backgrounds, Chou *et al.* in [6] and Ho *et al*. in [7] suggested random linear network coding, which is our target and is conceived to be the most practical scheme for single multicast flow cases. Lun *et al.* showed the utility of network coding on wireless network systems in [8], until then, researches of network coding were focused on wired networks. Katti *et al*. proposed practical solutions for wireless networks with multiple unicast flows in [9] and Park *et al.* suggested a practical protocol based on network coding for ad hoc multicasting networks and showed improvements of reliability of ad hoc network systems by network coding in [10]. In addition, using network coding in P2P was first proposed in [11] and recent feasibility studies on network coding in real testbeds have been done in [12] and Lee *et al*. showed the utility of network coding in mobile P2P systems [13]. Gkantsidis *et al.* also showed that network coding allows smooth, fast downloads and efficient server utilization on a P2P setting [3].

Shojania *et al*. suggested parallelization of network coding in [4]. They employed hardware acceleration into the network coding and used a multi-threaded design to take advantages of multi-core systems. There are some other performance enhancement techniques (e.g. [14], [15]). Their work is different from our work in

that their focus is reducing the computational complexity of encoding/decoding operation and ours focuses on improving decoding performance via parallelization.

There are many researches such as parallelization of matrix inversion [16], parallel LU decomposition [17], and parallelization of Gauss-Jordan elimination with block-based algorithms [18]. In fact, those existing parallel algorithms could be used to decode received packets of network coding. However, these algorithms need to receive the entire matrix before starting decoding operations.

In network coded systems, waiting for the entire matrix to be formed is not an optimal solution. In P2P settings, transfer units are delivered one by one and the time gap between the arrivals of transfer units can be large. Thus, instead of waiting all the packets to arrive, partial decoding is performed on reception of each transfer unit hence the name of "progressive" decoding [4]. Our focus is on this type of progressive decoding.

To enhance the performance of the progressive decoding, *Parallelized Progressive Network Coding (PPNC)* is proposed [4]. It is basically a variant of the Gauss-Jordan elimination algorithm. A simple description of Gauss-Jordan elimination borrowed from [4] is presented in Table 1.

**Table 1**. Operation of Each Stage in Progressive Decoding [4]

| ***Stages*** | Task Descriptions |
|---|---|
| ***A*** | Using the former coefficients rows, reduce the leading coefficients in the new coefficient row to 0. |
| ***B*** | Find the first non-zero coefficient in the new coefficient row |
| ***C*** | Check for linear independence with existing coefficient rows |
| ***D*** | Reduce the leading non-zero entry of the new row to 1, such that result in REF |
| ***E*** | Reduce the coefficient matrix to the reduced row-echelon form |

To enable progressive decoding, the stages of *PPNC* start operating when the destination receives a transfer unit containing coded packet and coefficient, that means a new row is added to matrix. On each transfer unit's arriving, the operations from *Stage A* to *Stage E* operate on the coefficient and packet matrixes to form the reduced row-echelon form. In these stages, *Stage A* and *E* are dominant procedures. According to [4], *Stage A* has 50.05%, and *Stage E* has 49.5% of decoding workload. So the parallelization is focused on *Stage A* and *E*.

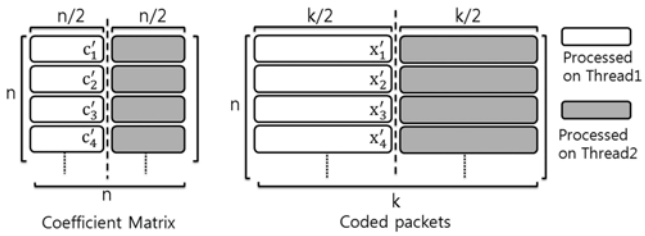



**Fig. 3.** Concept of Thread Dividing in *PPNC*

The main concept of the parallelization is to divide the coefficient matrix and packet matrix into a limited number of operational regions each of which is fed to parallel tasks (or threads). The regions are divided by vertically and equally as Fig. 3 with *PPNC*. Since dependency between threads exists, at start and end of each stage, synchronization between threads is needed.

# 3 Algorithms for Parallelization of Network Coding

In this section, we present an arithmetic analysis on the workload balancing problem of the parallel progressive decoding algorithm and propose three new parallelization methods. In this paper, we focus on the parallelization of *E* stage for the purpose of clearer presentation. Unless otherwise specified, other stages such as Stage A are parallelized with the same techniques used in [4].

## 3.1 Arithmetic Analysis of Thread Balancing

The best way to divide overall workload in parallel algorithms is to allocate same amount of load to each parallel task so that all the tasks such that they can start and end simultaneously. If the workload is unbalanced, the benefit of parallelism diminishes, which limits the performance of *PPNC* proposed in [4].

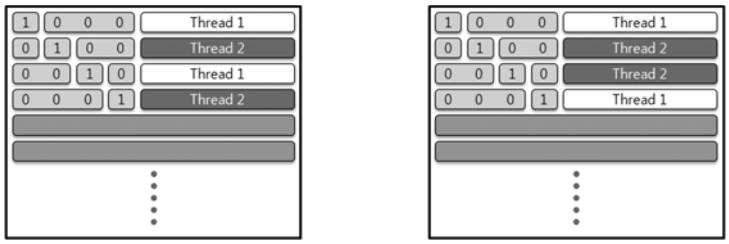
To illustrate the problem, let us assume that the size of coefficient matrix is $n \times n$. The *Stage E* operations start with all threads, but later, when index of decoding go to row of $\frac{n}{number\ of\ threads}$ , the first thread has no work during coefficient matrix operation on *Stage E*. The region for that thread is already filled with 0 and 1, and no additional operation is needed. If there are two threads, the first thread has no operation after $\frac{n}{2}$ row's operation. In case of 4 threads, first thread has no operation after $\frac{n}{4}$ row's operation. The more threads are added, the more inefficiency occurs.

To compute the workload of each thread, we define a sequence of a subtraction after a multiplication on a spot of matrix which operates in *Stage E*, to a unit operation. With arithmetical approach, in case of 2 threads, the first thread operates $\frac{n^3+6n^2+8n}{48}$ unit operations and the second thread operates $\frac{n^3+6n^2+8n}{48}+\frac{n^3+2n^2}{8}$ operations. The gap between two threads' numbers of operations is $\frac{n^3+2n^2}{8}$, and it is bigger than first thread's whole operation numbers. In case of 4 threads, the gap between the threads is getting larger. The first thread operates $\frac{3n^3+6n^2+16n}{192}$ unit operations, and the last thread operates $\frac{3n^3+6n^2+16n}{192}+\frac{3n^3+6n^2}{32}$ unit operations. The gap between the first and the last thread's operation numbers is $\frac{3n^3+6n^2}{32}$ in this case.

From the analysis above, we can easily see that the workload imbalance gap of *PPNC* is proportional to $n^3$. That is, using PPNC threads may have to wait for other treads possibly for a long time. We solve this workload imbalance problem and suggest more efficient parallel decoding algorithms in the next section.

### 3.2 DOA: Dynamic Operation Assignment for Balanced Workload

We suggest three different methods for efficient parallelization of the network coding. The first two methods are based on the horizontal division of matrix in order to balance the task separation. The easiest way of horizontal dividing is *round robin* (*RR*) method, which means a row is assigned to a thread, and the next row is assigned to the next thread, and continues as Fig. 4-(a). However, when two threads are assigned to any odd numbers of rows, the last row is always assigned to the first thread unevenly. Therefore, the first thread has the heavier workload, and unbalanced work distribution is made. More efficient horizontal dividing is *backward round robin* (*BRR*) after round robin as Fig. 4-(b). In this way, the thread of heaviest workload is changing, but also in case of operations on rows with odd numbers, perfect load balancing is not possible. Moreover, this method needs more complex arithmetic operations to find the row to operate than *round robin* and need more time for that. So we expect there exist some trade-offs; we will discuss this matters with the experimental results of two algorithms on real machines in Section 4.




(a) *RR - Round Robin* Method (b) *BRR – Backward Round Robin* Method

**Fig. 4.** Concept of Horizontal Separation for Balanced Task Partitioning

In case of round robin, the first thread, which has the maximum workload, would take $\frac{2n^3+9n^2+10n}{24}$ unit operations on 2 threads, and $\frac{2n^3+17n^2+30n}{48}$ on 4 threads. On the other hand, last thread which has the minimum load would take $\frac{2n^3+3n^2-2n}{24}$ on 2 threads and $\frac{2n^3-n^2}{48}$ operations on 4 threads. The gaps between threads are $\frac{n^2+2n}{4}$ with 2 threads and $\frac{3n^2+5n}{8}$ with 4 threads. Compared to the gap calculated with *PPNC*, we can find out that *round robin* is much more efficient.

The third method is to use dynamic vertical separation of operation area which is named *DOA (Dynamic Operation Assignment*). In this method, the dividing point is dynamically varied for each row operations as illustrated in Fig. 5. When the first row is to be handled as shown in the left most diagram, each thread is assigned $\frac{n-1}{2}$ columns, and when working on the second row and the third row shown in the next two diagrams, each thread works on area of $\frac{n-2}{2}$ and $\frac{n-3}{2}$ columns, respectively. If

the number of columns to be assigned is not the multiples of the number of threads, the remaining columns are assigned unevenly.

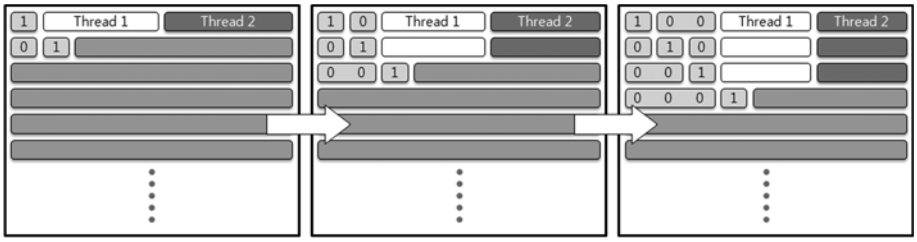



**Fig. 5.** Concept of Dynamic Thread Separation on Operation Area

As the algorithms progress, the region assigned to each thread is getting narrower and deeper. In this way, we can easily achieve fair balancing of workloads among threads. Imbalanced distribution happens when the number of remaining columns is not a multiple of *k* (where *k* denotes the number of threads). However, that kind of imbalance is negligible especially when *k* is smaller compared to *n* and the *DOA* is an efficient algorithm for parallelization of *Stage E*.

## 4 Experimental Results and Performance Analysis

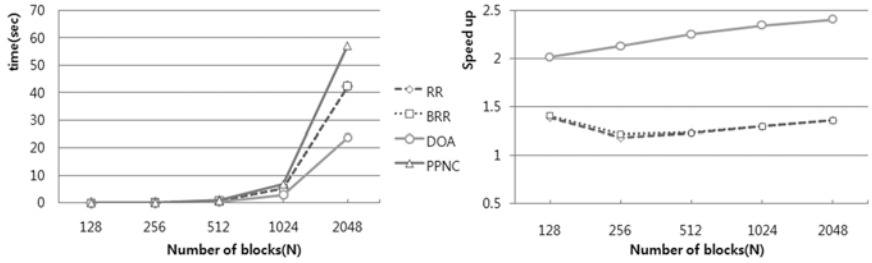
**Table 2.** Experimental Environment

| | **Dual-Core** | **Quad-Core** |
|---|---|---|
| CPU | Intel Core 2 Duo E6750 | AMD *Phenom-X4* 9550 |
| CPU Clock | 2.66GHz | 2.2GHz |
| RAM | 2GByte | 4GByte |
| Cache Configuration | 2x32KByte L1 cache<br>4MByte Shared L2 cache | 4 x 128KByte L1 cache<br>4 x 512KByte L2 cache<br>2MByte Shared L3Cache |
| Operating Systems | Fedora Linux Core 8 | Fedora Linux Core 8 |

In this section, we evaluate the proposed three algorithms via extensive experiments on real multi-core machines. The specification of the machines we have used for our experiments is described in Table 2.

### 4.1 Performance Evaluation Considering *Stage E* Only

The first set of experiments is carried on in order to see the performance of Gauss-Jordan elimination only with the coefficient matrix; which means we exclude the execution time spent on the packet matrix.

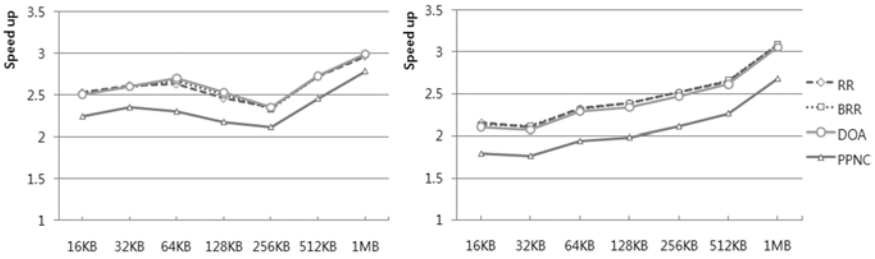



(a) (b)

**Fig. 6.** Execution Time on *Stage E* and Speed-up (Coefficient Matrix Operation Only)

Fig. 6-(a) shows the execution time spent only on *Stage E* of the coefficient matrix operation for four different algorithms: *PPNC*, *RR*, *BRR*, and *DOA*. Fig. 6-(b) presents the speed-up of *RR*, *BRR*, and *DOA* compared to *PPNC*. The size of the file used is 1MB. From the figures, we notice that *DOA* shows the best performance. The speed-up factor ranges from 2 to 2.4 compared to *PPNC* (again, when considering only *Stage E* of the coefficient matrix operation). We also notice that the execution time of *round robin (RR)* and *backward round robin (BRR)* are very similar. This is due to the fact that the advantages/disadvantages of two different thread assignment methods diminish in the real implementation.

### 4.2 Speed-Up Comparison on Dual-Core Systems




(a) (b)

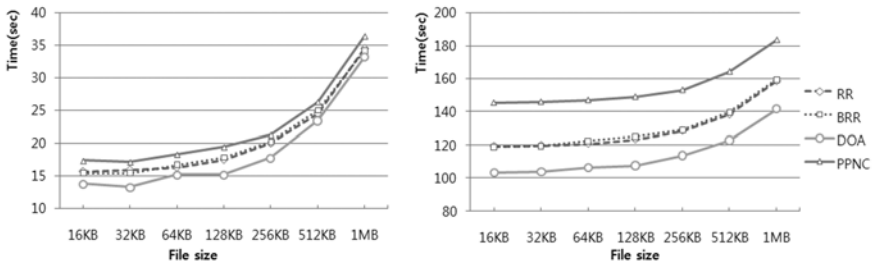
**Fig. 7.** Decoding Process Speed-up with Varying the Data Size

In this part, the speed-up factor of total decoding time considering a whole file is measured. We have calculated the speed-up factors compared to sequential algorithms for various experimental scenarios. In Fig. 7, we can find out the speed-up factors of decoding process with four proposed algorithms on a dual-core processor. Fig. 7-(a) and (b) show speed-ups with $n$ = 1024 and $n$ = 2048 on a dual-core processor. On the

dual-core processor, *RR*, *BRR*, and *DOA* do not show sharp improvement on speed-up while showing a better performance compared to *PPNC*. These results prove that our proposed algorithms are more efficient than *PPNC*.

### 4.3 Total Decoding Time Comparison

In this part, the total execution time on process is measured. In Fig. 8, we can find out the decoding time of the various file sizes. Fig. 8 presents the execution time in case with $n$ = 1024 at (a), and $n$ = 2048 at (b). Due to the *PPNC*'s unbalanced parallel workload distribution, the *PPNC* approach results in the longest decoding time compared to the other three algorithms in the whole decoding process. In fact, the operations in the remaining stages but *Stage E* in our algorithms are very similar to those operations in PPNC. In other words, the speed up in Stage E is a dominant factor.




(a) On Quad-Core, n = 1024 (b) On Quad-Core, n = 2048

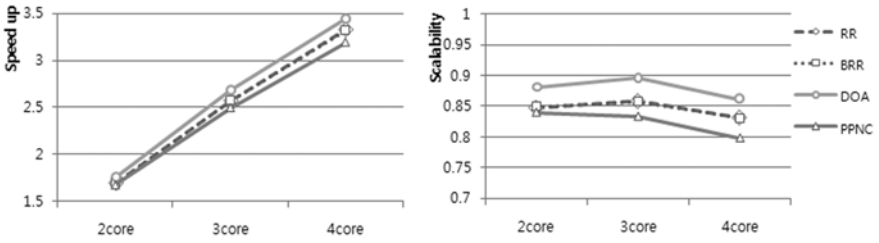
**Fig. 8.** Decoding Execution Time (in sec) with Different Data Sizes on Quad-core

Ratio of execution time on coefficient matrix increases with larger coefficient matrix size, therefore *DOA*, *RR*, and *BRR* show better performance results on whole decoding process with large $n$. It is very important finding since the performance improvement in large sizes is crucial for file swarming as mentioned in Section 2.1. As indicated in [4], the task of coding more than 128 blocks is challenging and should be addressed. We claim that our approach can provide a better solution for the network coding with large numbers of blocks.

### 4.4 Scalability Comparison

We measure the speed-up factors and scalability using various numbers of cores to verify the efficiency of each algorithm. The scalability means the capability to accelerate the operation speed with the addition of cores. In this section, the results on scalability are derived with the speed-up divided by the number of cores. All the results are calculated considering the execution time of a sequential program on a single core. Fig. 9-(a) shows the speed-up factors of each algorithm with using

various numbers of cores. We can find out the *DOA* shows the best performance over other three algorithms. Fig. 9-(b) shows the scalability calculated from the results shown on Fig. 9-(a). It is also demonstrated that the *DOA* algorithms show the best scalability and these results prove the efficiency of *DOA* in multi-core environments.




(a) Speed-Up Factors (b) Scalability

**Fig. 9.** Speed-up Factors and Scalability of Each Algorithm (2MB Data, $n$=1024)

## 5 Conclusion and Future Work

This paper introduced efficient parallel algorithms for the random network coding. To be more specific, we proposed "balanced" parallel algorithms. We showed via analysis that our algorithms have less workload difference between tasks compared to the previously proposed *PPNC* algorithm. Via real machine experiments, we showed that our algorithms achieved speed-up of 3.05 compared to a sequential implementation and showed 14~30% improvement over PPNC. Moreover, our algorithms showed better scalability than *PPNC* on the number of processing units (e.g., processor cores). We expect that our work can be applied to further enhance the performance of various network coding applications such as peer-to-peer file sharing systems.


### Acknowledgement

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (KRF-2008-313-D00871).


## References


1. Geer, D.: Industry trends: Chip makers turn to multi-core processors. Computer, vol. 38, no. 5, pp. 11--13 (2005)
2. Ahlswede, R., Cai, N., Li, S.-Y.R., and Yeung R.W.: Network information flow. IEE E Trans. Inform. Theory, vol. 46, no. 4, pp. 1204--1216 (2000)

3. Gkantsidis, C., Rodriguez, P.: Comprehensive View of a Live Network Coding P2P System. In: IMC'06, Rio de Janeiro (2006)
4. Shojania, H., Baochun Li: Parallelized Progressive Network Coding With Hardware Acceleration. In: 15th IEEE International Workshop on Quality of Service, pp.47--55, (2007)
5. Koetter, R., M´edard, M.: An algebraic approach to network coding. IEEE/ACM Trans. Networking, vol.11, no.5, pp. 782--795 (2003)
6. Chou, P., Wu, Y., Jain, K: Practical Network Coding. In: 51st Allerton Conf. Comm un.,Control and Computing (2004)
7. Ho, T., M´edard, M., Koetter, R., Karger, D.R., Effros, M., Shi, J., Leong, B.: A R andom Linear Network Coding Approach to Multicast. IEEE Trans. Information Theo ry, vol.52, no.10, pp.4413--4430,
8. Lun, D.S., Ratnakar, N., M´edard, M., Koetter, R., Karger, D.R., Ho, T., Ahmed, E., Zhao, F.: Minimum-cost multicast over coded packet networks. IEEE Trans. Infor m. Theory, vol. 52, no. 6, pp. 2608--2623 (2006)
9. Katti, S., Rahul, H., Hu, W., Katabi, D., M´edard, M., Crowcroft, J.: XORs in the Ai r - Practical Wireless Network Coding.
   IEEE/ACM Transactions on Networking. vol. 16, no. 3, pp. 497--510 (2008)
10. Park, J.-S., Gerla, M., Lun, D.S., Yi, Y., M´edard, M.: Codecast: a network coding bas ed ad hoc multicast protocol. IEEE Wireless Communications, vol.13, no.5, pp.76--81 (2006)
11. Gkantsidis, C., Rodriguez, P.R.: Network coding for large scale content distribution. In: 24th Annual Joint Conference of the IEEE Computer and Communications Societies. vol.4, pp. 2235--2245 (2005)
12. Wang, M., Li, B.: Lava: A Reality Check of Network Coding in Peer-to-Peer Live Streaming. In: INFOCOM 2007. 26th IEEE International Conference on Computer Co mmunications. IEEE, pp.1082--1090 (2007)
13. Lee, U., Park J.-S., Yeh, J., Pau, G., Gerla, M.: CodeTorrent: Content Distribution u sing Network Coding in VANETs. In: 1st international Workshop on Decentralized R esource Sharing in Mobile Computing and Networking. MobiShare '06. ACM (2006)
14. Ma, G., Xu, Y., Lin, M., Xuan, Y.: A content distribution system based on sparse li near network coding. In: NetCod07, 3rd Workshop on Network Coding, Miami (2007)
15. Maymounkov, P., Harvey, N.J.A., Lun, D.S.: Methods for efficient network coding. Allerton, Monticello (2006)
16. Csánky, L.: Fast Parallel Matrix Inversion Algorithms. SIAM J. Computing, vol. 5, pp. 618--623 (1976)
17. Bisseling, R.H., Van de Vorst, J.G.G.: Parallel LU decomposition on a transputer network, LNCS, vol. 384, pp. 61--77 Springer Berlin, Heidelberg (1989)
18. Melab, N., Talbi, E.-G., Petiton, S., A Parallel Adaptive Gauss-Jordan Algorithm, Th e journal of supercomputing, vol. 17, no. 2, pp. 167--185 (2000)