Access Complexity: A New Complexity Analysis Framework for Parallel Computation

Daisaku Yokoyama[1] and Takashi Chikayama[1]

The University of Tokyo, Tokyo, Japan

Abstract. Recent advances in computer systems have been making the conventional computational complexity theories inappropriate in various situations. The random access memory (RAM) model was made unrealistic by the large speed gap between the processing units and the main memory systems. Distributed computing environments have obsoleted most of the parallel computation models proposed so far due to non-negligible diversity in communication delay. In this paper, we propose a new framework for computational complexity, named *access complexity*, in which the cost is assumed to lie in data transfer rather than computation itself. The model tries to capture all levels of system hierarchy, from cache systems to globally distributed environments. The abstract machine provided by the model has a uniform memory field and a uniform communication layer, which are meant to capture a variety of parallel computation environments. Diverse access costs in the actual computing systems are modeled as different latencies in the communication layer in a uniform way with a simple congestion model. With the model, some parallel algorithms, bitonic sort, merge sort, and FFT, namely, are analyzed, showing that the model facilitates performance analysis. We also show the appropriateness of the model through comparing the predicted and the measured performances of these algorithms.

1 Introduction

1.1 Background

Traditionally, the random access memory (RAM) model [1] that assumes a flat memory system with a fixed constant access time has been used as the standard computational complexity model. The assumption, however, has become unrealistic with recent computer. Modern architectures have a deep memory hierarchy, and the speed gap between these memory levels has been and still is rapidly widening.

Several studies that do not assume constant memory access cost have been made. Aggarwal *et al.* proposed the Hierarchical Memory Model (HMM) [2], which represents the access cost as a function of memory location. In this model, the access cost of data at address x is proportional to $\log x$. Data that have small addresses are located on small but fast memory, and can be accessed with small cost. If we want to handle much larger data, these data should be

located on large memory addresses, which corresponds to large but slow memory. Alpern *et al.* proposed the Uniform Memory Hierarchy Model (UMH) [3], which aims to represent the same behavior in another way. This model has a series of nested memory elements. Their sizes, latencies, etc. are defined recursively with the same ratio. Although these two models can be used to figure out the behaviors of many sequential algorithms, they cannot be applied to analyze parallel algorithms.

1.2 Related Work

When considering parallel computational complexity, LogP model proposed by Culler *et al.* [4] is widely known. In this model, a parallel computational environment is represented as computers connected by a uniform communication channel characterized by four parameters. In the real world, however, the communication layer have become nonuniform with large-scale distributed computation environments that are getting popular these days. Remotely located computers have different communication latency. That difference inevitably affects the program execution time, and thus the gap between LogP model and the real world has been widening. There are studies that add some parameters to the LogP model aiming to describe the behavior of the communication channel more precisely. However, while getting fitness for a certain distributed architecture and certain parallel algorithm, these models lose generality and simplicity instead. Moreover, these models are for only communication links, and we have to combine it with another model of local computation.

The coarse grained multicomputers model (CGM, proposed by Dehne *et al.*) [5] and bulk-synchronous parallel model (BSP, proposed by Valiant) [6] are also well-known parallel computational complexity models. In these models, computers with a certain amount of local memory are connected over a communication network with a certain topology. Parallel algorithms are modeled as repeats of two phases: the "independent computation phase in each computer" and the "global communication phase throughout the all computers". The computational cost of the whole algorithm is basically the sum of the costs of these two phases that are considered separately. Williams *et al.* revised BSP to model the differences of capacities of communication links or computational powers at each location and obtained the heterogeneous BSP model [7]. The communication and computation phases in this model are synchronized throughout the whole computation environment, which is the same as in the BSP model. Heterogeneities are modeled by the fact that the part with the poorest communication or computation capacity limits the duration of a synchronization phase. The communication cost in this model is defined by the bandwidth of interfaces of each computers and the size of data. This modeling is quite simple, but does not consider the network topology or locality of communication.

The model that handle "computation in one processor" and "communication between processors" separately demands for combining two computational models to analyze algorithms. Unfortunately, analysis in this way may become

awkward. Indeed we may be able to find a set of many parameters of this combined model precisely that represents certain architecture, that is to say, an environment with a certain memory hierarchy and a network topology. These parameters, however, should be changed when the same algorithms are analyzed on another architecture. The trend of computer architecture is changing continually, thus these models need to be changed frequently to follow the trend. This means that we cannot discuss the merits and demerits of a certain algorithm itself.

When we use enormously many computers such as in a GRID, treating each communication links separately becomes quite hard. That assumption, namely, "parallel computation environment consists of some computers connected by a communication network", makes the computation model very complex, and not applicable to large-scale distributed computation getting more and more popular these days.

In 2003, Rauber *et al.* proposed a simple measure derived from the profile of memory accesses of an algorithm, called Locality Measure [8], to explain the performance of parallel algorithms being independent of the execution environment. Our model aims at the same direction. While their Locality Measure tries to extract the qualitative measure indirectly by the profiling actual computation, our model tries to explain the behavior directly by being aware of the structure of computers.

1.3 Proposal

We reconsider computational complexity based on the idea that communication is where the computational cost lies, not in arithmetic and logical operations. The computation time increased by the memory hierarchy and the network topology are treated as different levels of the same aspect, that is, the communication latency of data transfer.

The proposed computational complexity framework ignores the distinction between inside and outside of a single computer, between computation in a computer and communication on the network. In this framework, computers in the real world are modeled as a uniform single abstract machine, ignoring the above distinction. This model assumes that the distance x can be defined between two memory locations, and the communication cost between them can be defined as a single function $f(x)$. The behavior of any memory hierarchy and any network topology can be described by this $f(x)$, in a simple and uniform way. This uniform function leads to easy estimation of the computational cost when we design an algorithm.

This model aims to be the general measure that predicts the common behavior among diverse parallel computation environments, rather than the exact computation time in certain environments. Indeed, there are some jumps of access latencies in certain hierarchical memory, and these jumps are specific to each architecture. The latency gap between accessing local memory and communicating with another computer is considerably large when we try to figure out

the precise behavior on a certain architecture. We dare to ignore these details, however, and approximate these gaps simply by a function $f(x)$.

We propose this model, called *Access Complexity*. In this paper, we show the concept of this model, computation on this model, and the design of the abstract machine. We also show the appropriateness of the model, through analyzing some parallel algorithms.

2 Computation Model

2.1 Basic Concepts

The access complexity model assumes an abstract architecture with the following basic concepts.

1. In this architecture, there is a single continuous memory space with finite density. This memory space is currently one dimensional.
2. The processing capability exists at every memory location. Each arithmetical or logical operation is performed at a certain location. We assume that a *computation entity* executes the operations. The computation entities can be at every memory location, and there can be infinitely many of them at the same time.
3. The program execution time is dominated only by the memory access time. Operation time is simply ignored. This does not lead to infinitely fast processing. Any operation requires fetching of its operands and storing of its result. Fetching and storing require data transfer and thus with some costs. Data accesses are through a communication channel over the memory space.
4. The cost of memory accesses depends on the distance between two memory locations, the point that data is stored and the point that the computation entity exists. An access order to a data placed at a location P located at the distance of x from the location of the computation entity reaches after the time period of $f(x)$. Constant time l is required in accessing at location P. In addition, another $f(x)$ time is needed for the read data or write acknowledge to reach the original computational entity. Therefore, the total memory access cost is $2f(x) + l$.
5. The function $f(x)$ increases monotonically, with the restriction $f(0) = 0$. $f(y) < f(x) + f(y - x)$ is true under the condition $0 < x < y$, in other words, $f(x)$ is a concave downward function. This means that the communication latency cannot be shorten by relaying it at any location.
6. We try to describe the effect of concentration of memory access by the congestion in the communication channel. Without this limitation, infinitely many memory accesses to the same data can be made at the same time, which is quite far from the reality.
7. The computation entities only have its location and its program counter (PC). They do not have any extra memory such as registers. They can change their locations on the memory freely. Moving a computation entity needs the time $f(x) + l$, just like memory accesses. Moves of a entity cannot overtake

a data transfer. Increasing the PC does not need any cost, that is, the cost of fetching instructions of a program is ignored in our model.

8. The communication channel have a certain capacity. A communication that exceeds the capacity gets a penalty of latency. This restricts the parallelism of the whole algorithm. Although infinitely many computation entities can be there, the execution time of the whole algorithm cannot be shorter than a certain lower bound due to this congestion penalty.

Let us try to describe a single computer that has one CPU in our model. This environment is represented as the memory space with only one computation entity. The memory area neighboring the location of the computation entity represents the registers; they have the shortest distance from the computation entity and thus with the shortest latency. The area next to this corresponds to the L1 cache memory with a bit larger distance, next to it is the L2, and then comes the main memory, and so on. The capacity of memories of the real machine becomes larger as the latency becomes longer. We can emulate the communication behavior of the real machine by defining the function $f(x)$ so that relation of capacity and latency is represented well. In many cases, the memory capacity grows exponentially proportional to the latency. The study of HMM [2] concluded that the setting $f(x) = \log x$ is generally appropriate for diverse single computers. The setting of $f(x)$ is still open to discussion due to the rapid increase of relative latency today.

In modern architecture, data in the main memory located close to recently accessed data are automatically moved to the cache memory, and can be accessed faster after that. In access complexity model, however, this cache mechanism should be explicitly described by the algorithm designer. There are two ways to emulate the cache behavior in our model. One is to move the data explicitly; to move the required data far away to a location closer to the computation entity. The other is to move the computation entity itself.

A system with distributed computers is described with the computation entities of the same number as the number of computers. The memory location of our model defines the computer where the data really is. Note that the owner of data is not a computation entity. In this situation, moving a computation entity represents not only the memory hierarchy, but also the remote method invocation. When we consider about a SMP machine, that environment is represented in the same way as distributed systems, with different mapping of the abstract and the physical memory.

In our model, the memory is currently a one dimensional array. This widens the gap between the model and the real environment. In the model of SMP, for example, let us consider three computation entities communicating one another. In our model, two entities at the both end communicate over a longer distance than the center entity, and thus are concluded to show lower performance than the centered one. Of course, this conclusion does not match the real world. The reverse situation is found in distributed computing. Some network topologies may have multiple communication channels between two computers, or may have asymmetric channel where the latencies depend on the directions. These

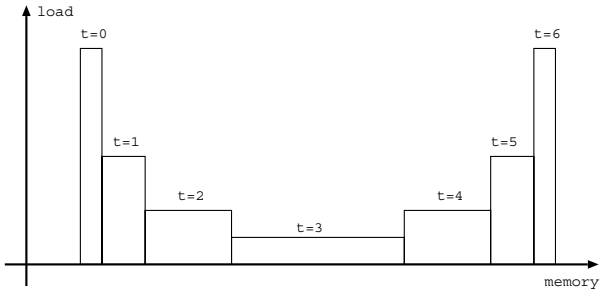

Fig. 1. Transition of communication load caused by a packet

facts cannot be represented with the one-dimensional memory model. Extending the memory model to have multi-dimensional or more complex topologies and appropriately defining the distance and communication may fix the problem. The definition of the distance, however, will be much more complicated in such models. Furthermore, there may be some detours to avoid congestion with multi-dimensional communication channel, that makes analysis of the communication quite hard. Topologies of actual computation systems may differ considerably. It would be quite difficult to find a common abstract topology that can explain such systems precisely. As we wanted to build a common framework for performance analysis of parallel algorithms, we decided to adopt the simplest possible one-dimensional memory model.

2.2 Communication Channel

The computational cost of our access complexity model is greatly affected by the model of communication channel. The communication channel is characterized by the function $f(x)$, as we discussed above. Currently we adopt the assumption that $f(x) = \log x$. According to the study of HMM, this definition explain the memory hierarchy of single computer at a certain amount. We assume that this function also describe the network latencies of general distributed environment, with continuous growth rate to the latency of local memory.

To consider the locality of communication, we try to model the communication with the moving packet. The travel time of a communication packet to its destination follows the latency function $f(x)$. Figure 1 shows a behavior of a traveling packet. The packet is initiated at time $t = 0$, and then moves to the adjacent location, that is the location with distance 1, at $t = 1$. The packet speed increases exponentially until the halfway point between start and goal point, at $t = 3$. After the halfway point, the packet speed decreases exponentially, symmetrically to the first half. This increasing and decreasing process of the packet speed leads the total travel time of $O(\log x)$ for the distance x.

A packet burdens a certain load to a certain area around its position of the communication channel. The transition of the load of a packet is as shown in

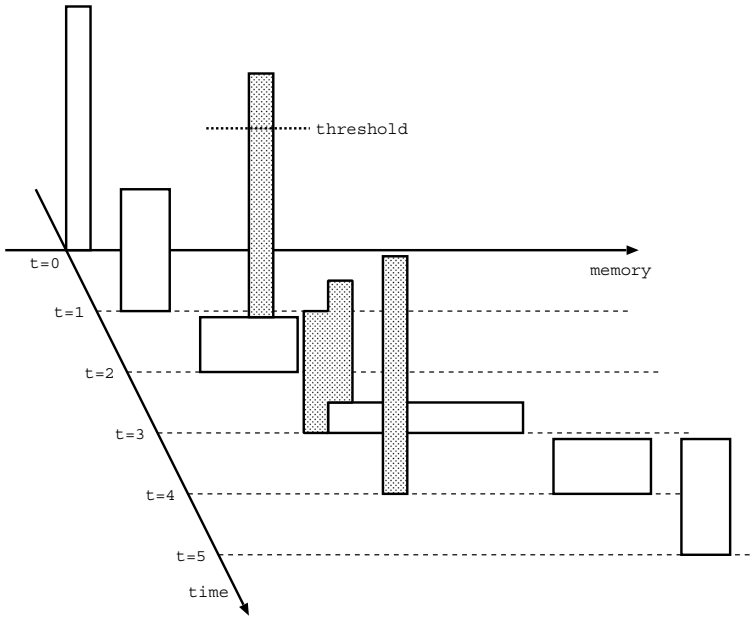

Fig. 2. Transition of communication load caused by multiple packets

Fig. 1. The area that a packet burdens is defined according to the packet speed. One packet has a constant load at a time; with the load of a packet at a certain location x denoted as $l(x)$, the equation

$$\int l(x)dx = constant \tag{1}$$

is always true. That is to say, a packet with high speed affects a wide area, but its load is thin. The $l(x)$ function with this constraint may vary, but we adopt a simple function for the discussion below. When the packet speed at a certain time is v, it burdens the area of width v with the uniform load of $1/v$.

A packet consumes up a part of the limited throughput of the communication channel with its load. If two packets burden the same location, then the load of the two packets at that location are summed up, as shown in Fig. 2. In this example, two packets interfere each other. Two packets both moving to the right are generated at $t = 0$ and $t = 1$. At time $t = 2$, 3 and 4, two packets occupy at the same location, the loads of them at that location are summed up. If the load exceeds the throughput of the channel at any point, the transfer of all packets on the channel are slowed down. If the load is two times the throughput of the channel, the communication delay twice as large as that of a channel with vacancy.

This model neglects the layered structure of the communication channel of the real world. For example, a data packet of a local area network interferes with

the communication of main memory bus and slows down the computation to a certain degree. However, this model very much simplifies the communication behavior and makes communication analysis much easier for some communication pattern.

3 Analysis of Parallel Algorithms

In this section, three well-known parallel algorithms, bitonic sort, merge sort, and FFT, are analyzed. Performance of programs based on these algorithms on the real machines are measured and compared to theoretical predictions based on (P)RAM model and the access complexity model.

3.1 Bitonic Sort

The bitonic sort algorithm forms a kind of sorting network. This algorithm repeatedly compares the given data in a regular manner. Pairs of data to be compared are fixed beforehand regardless of their contents, making it easily parallelized.

The program used in these measurements starts with the input data placed in an array whose elements are allocated contiguously in the memory. The program sorts out the data in the same array. Sorting of n elements consists of one bitonic merge process of n elements and two bitonic sort processes of $n/2$ elements. Bitonic merge of n elements is by repeatedly comparing the ith element and the $i + n/2$th element, increasing i from 0 to $n/2 - 1$. Thread partition is decided statically: With P threads, elements i assigned to the pth thread satisfy $np/2P \leq i < n(p+1)/2P$.

First, we figure out the computation cost with the RAM model.

The bitonic sort of n elements includes $(n/2) \times (\log n(\log n+1)/2)$ operations of comparison, and thus the computational complexity is $O(n(\log n)^2)$ with the RAM model. In parallel execution, all of $n/2$ comparisons can be executed concurrently, making the time complexity $O(n(\log n)^2/P)$ with P processors. With n processors, the lower bound of the time complexity is $O((\log n)^2)$.

Measured results of our experiments on the real machines are shown in Fig. 4. The measurement environments are:

- Ultra SPARC III Cu 1.2GHz, 8 CPU, and
- Pentium4 Xeon 2.4GHz, 2 CPU.

Program is parallelized with the pthread library, with shared memory. Four sequences correspond to executions with single and multiple threads on two machines. Eight threads are used on 8 CPU Ultra SPARC and two are used on Xeon processors. The horizontal axis shows the number of sorted data elements. The vertical axis shows the ratio of measured and predicted increase of execution time when the data size is doubled. The value of r at data size 2^k means $r = (m_k/m_{k-1})/(p_k/p_{k-1})$ where m_j and p_j are measured and predicted execution time with size 2^j. We call this value the *diremption factor* of the model at some

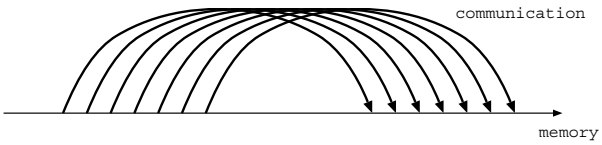

Fig. 3. Concurrent communication from and to continuous locations

data size. When the model appropriately explains the behavior, this value should converge to 1.

As we can see in the Fig. 4, the diremption factor keeps to be greater than 1 with the number of elements grows. As the memory hierarchy is ignored in the RAM model, increase of memory accesses with larger data size is not taken into consideration. This becomes apparent with large data sizes.

Now, we analyze the algorithm with the access complexity model.

In the access complexity model, layouts of data and computation entities affect the computational cost. The layout of the data can be specified in the program, but layout of the computation entities cannot. Thus, these location should be decided appropriately in the analysis. In this bitonic sort program, the memory cache mechanism works effectively, because memory accesses are continuous. Therefore, we have to take caching effects in consideration in the analysis. Here, we assume that the computation entity moves to the location of one of the two items each time comparison is made. With this setting, the other item is located constant distance away for all items in a single bitonic merge phase.

One comparison in the merge of n elements require communication at a location of distance $n/2$, and thus the operation should have latency of $O(\log n)$. By applying this recursively, the whole bitonic sort is concluded to have the computation cost of $O(n(\log n)^3)$.

In parallel execution with P processors, P communications of the same memory distance are generated at every $n/2P$ distances. The highest communication congestion occurs at $P = n/2$, with communications to the same direction and with the same distance P, located at the continuous P memory elements. Figure 3 shows this communication pattern.

In this situation, the load of communication channel at a certain location x consists of the packets of the same figure, all off by the distance one. The load is calculated by

$$\sum^{d} l(x-d)$$

with the load of a packet at a certain location x denoted as $l(x)$. With the equation (1), which means a packet has a constant load, the summed-up load is constant. Even when multiple packets are generated in this situation, the load does not exceed the load of only one packet there. That is, this communication pattern does not bring any congestion to the communication channel.

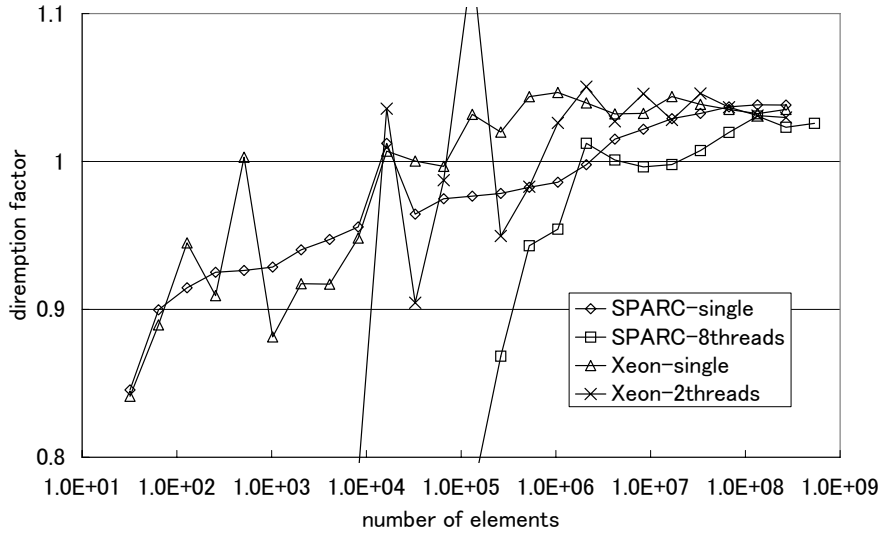

Fig. 4. Diremption factors of the RAM model in bitonic sort

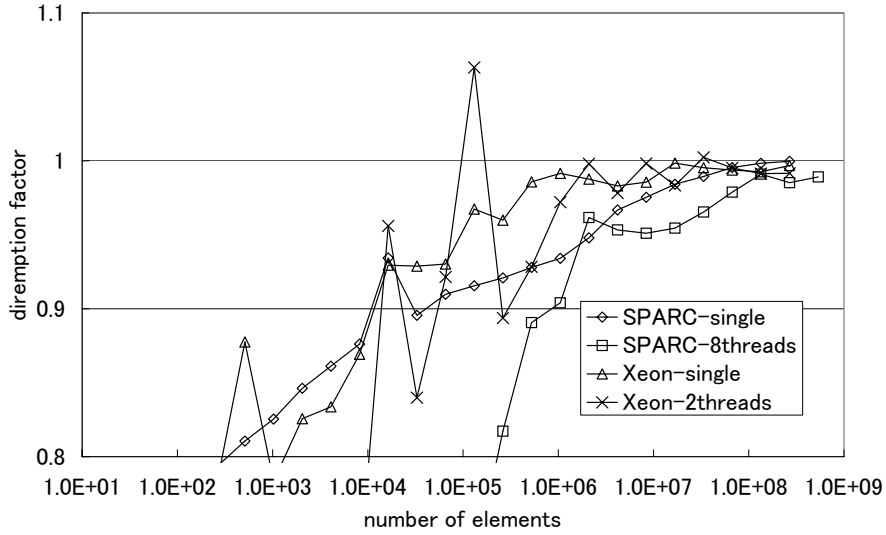

Fig. 5. Diremption factors of the access complexity model in bitonic sort

Therefore, parallel execution does not slow down, and the computational complexity is $O(n(\log n)^3/P)$, with the lower bound of $O((\log n)^3)$.

Diremption factors of the measured and predicted performance are shown in Fig. 5. The factor seems to converge to one with large number of data elements. That is, the access complexity model can estimate the computation complexity of bitonic sort more precisely than the (P)RAM model.

3.2 Merge Sort

The merge sort is also a deterministic algorithm, making the analysis easy. To sort n items of data, the program uses two arrays with n elements each contiguously allocated. To make a sorted sequence of size k, the program repeats comparing two elements of two sorted sequences of length $k/2$, then copy the appropriate element to a buffer of size k. The former, already sorted sequences, are in one memory array, and the later buffer is in the other. The two memory arrays are used alternatingly. The input data is placed in one of the two arrays.

With the RAM model, the computational complexity is $O(n \log n)$, and the time complexity with infinite number of processors is $O(n)$.

Let us analyze it with the access complexity model. We assume that the two arrays of size n are located adjacently. Communication is needed in two situations: When comparing two sorted part of size $k/2$, and when copying the result to the memory located at the distance n. Here, copying of the result has the dominant cost, as it requires more remote communication. As distances of element copying is affected by the initial order of the data, we have to consider the most time-consuming case. We concluded that copying takes the longest time when all the k elements are already sorted and every elements accesses to the location of distance n, as shown in Fig. 3. If any two elements of Fig. 3 are to be exchanged, communication latencies are changed from $f(n) + f(n)$ to $f(n-1) + f(n+1)$. Because of the convexity $f(x)$, $2f(n) > f(n-1) + f(n+1)$ holds. Therefore, the most time-consuming situation is as shown in Fig. 3, and the total communication cost is $k \log n$.

Under this condition, the comparison of two sorted part requires the communication cost of $2\sum_{i=1}^{k/2} \log i = k \log k - 2(k-1)$. With the computation cost of merge sort at the size k denoted as $T(k)$, the recurrence equation

$$T(k) = 2T(k/2) + k \log n + k \log k - 2(k-1)$$

holds. Solving this equation, the computational complexity is $O(n(\log n)^2)$.

Different data placement scheme can be considered, such as allocating presorted sequences and sorted sequence alternatingly. This placement, however, does not change the complexity concluded above.

The same analysis as in bitonic sort above concludes that the communication channel is never congested in parallel computation, Therefore, the parallel computational complexity is $O(n \log n)$, with enough number of processors.

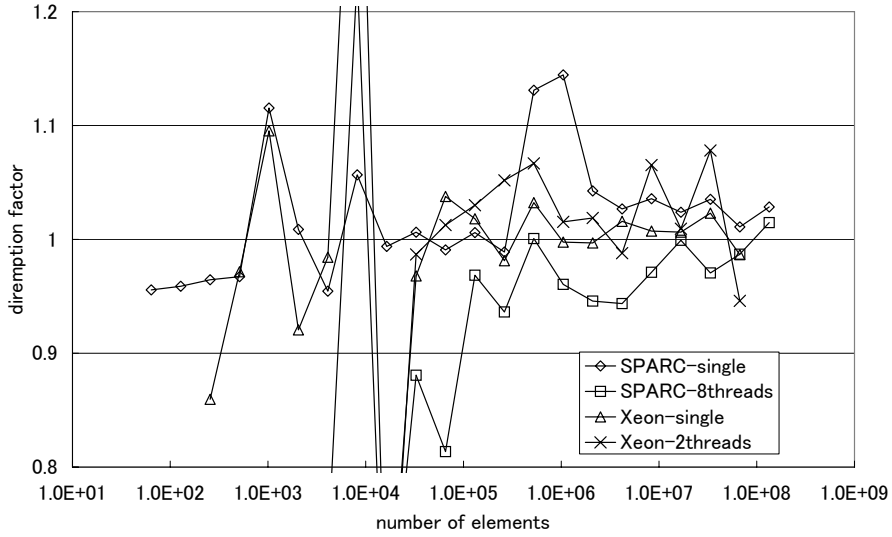

Fig. 6. Diremption factors of the RAM model in merge sort

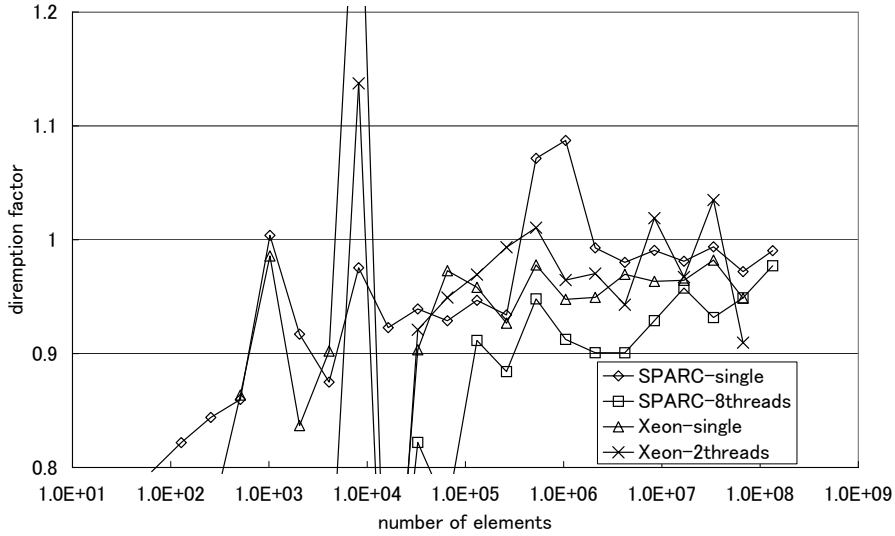

Fig. 7. Diremption factors of the access complexity model in merge sort

Diremption factors of measured and predicted performance with two complexity models are shown in Fig. 6 and 7, respectively. With single CPU SPARC and dual Xeon, the factor with the RAM model keeps to be above one, while it converges to one with the access complexity model. On the other hand, with single Xeon and 8CPU SPARC, the RAM model predicts better than the access complexity model. In general, the RAM model tends to underestimate the complexity while the access complexity model tends to overestimate it. That is, the merge sort program showed performance somewhere inbetween $O(n \log n)$ of the RAM model and $O(n(\log n)^2)$ of the access complexity model, and neither models can completely explain it.

The merge sort algorithm reads two contiguous memory areas for comparison and writes also to a contiguous memory area. This access pattern with high space locality fits the cache memory mechanism quite well. The program analyzed with the access complexity model does not explicitly represent the behavior of the cache mechanism. This might be the reason of the gap between the measured and predicted performances. As faithfully representing the cache behavior will considerably complicate the analysis, we are still looking for an algorithm that depicts the essence of the cache mechanism without much complexity.

3.3 FFT

Many FFT algorithms are known to improve the memory accesses locality compared to the naive FFT. There are also many methods to reduce the number of arithmetic operations. Although these refinements improves the performance by constant factors, they do not change the order, while making analysis harder, especially when the algorithm is parallelized. Therefore, we analyze the simplest FFT here, that performs "butterfly circuit $\rightarrow$ two FFT circuits of the size $n/2 \rightarrow$ shuffle circuit" with the input data of size n. The program uses two continuous memory arrays of size n, one for placing input data and the other for temporary working storage.

In the RAM model, the complexity is $O(n \log n)$, and the upper bound of it with parallelization is $O(\log n)$.

We analyze the algorithm with the access complexity model using two memory arrays placed adjacently. Let us consider the partial FFT of the size k performed during the FFT of size n. At the butterfly circuit, communications with latency $\log(k/2)$ occur k times. Then moving the data to the temporary buffer needs k communications with the latency of $\log n$. The shuffle circuit moves data from temporary buffer to the other, which takes the k times cost of $\log n$. Thus, with the computation cost of FFT of the size k denoted as $T(k)$, the recurrence equation is as follows.

$$T(k) = 2T(k/2) + k \log(k/2) + 2k \log n$$

Solving this equation, the computational complexity is $O(n(\log n)^2)$. Behavior of parallel computation can be discussed in the same manner as that of bitonic sort, and the complexity is concluded to be $O((\log n)^2)$ with enough number of processors.

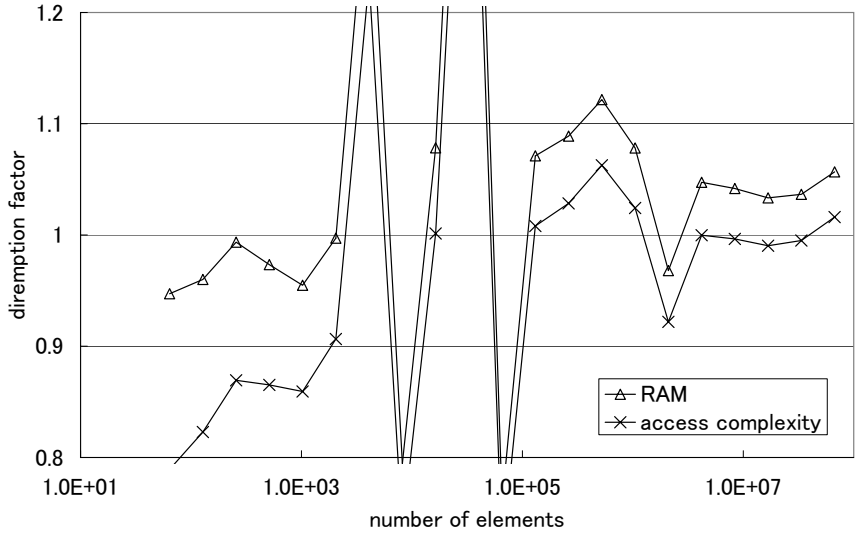

Fig. 8. Diremption factors of two models in pseudo FFT

The results of a preliminary experiment is shown in Fig. 8. This program does not actually perform the FFT; instead of calling trigonometric functions in the butterfly operation, it simply multiplies a constant. Calling trigonometric functions provided by commonly used libraries are computationally too heavy, dominating the execution time, and the effect of memory accesses will be hidden under it. High performance FFT programs use pre-computed tables to accelerate the computation of trigonometric functions, but the size of table such tables significantly affects the performance. To avoid this complication, we ignored this part of the algorithm. This program is not parallelized, and the evaluation environment is Opteron 1.4GHz. As we can see in Fig. 8, the diremption factor with the RAM model keeps to be above 1, while that of the access complexity model converges to it.

In contrast to the merge sort, this simple FFT program, the shuffle circuit accesses the memory almost randomly, hence the cache mechanism does not work well. This was probably a reason that prediction with better precision could be made with a model totally unaware of the cache mechanism.

3.4 Experimentation of Congestion in Communication Channel

As discussed in section 3.1, the performance of bitonic sort algorithm on shared memory is not affected by communication congestion. Although this is as predicted with the access complexity model, we should confirm this conclusion in en-

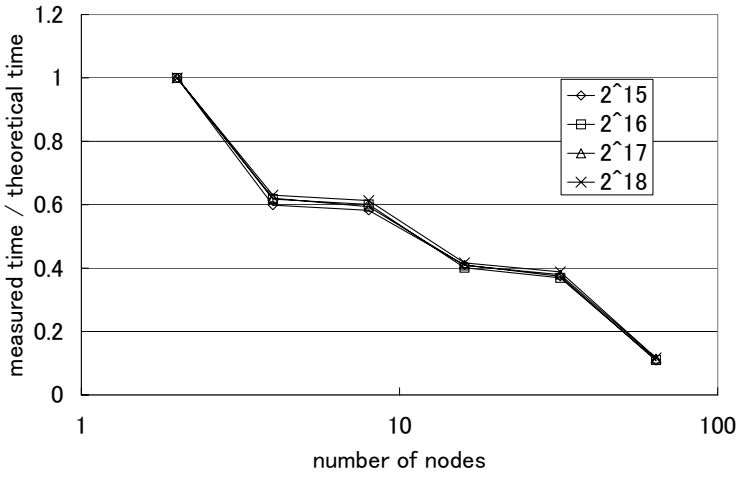

Fig. 9. Ratio of measured and predicted performances in parallel communication

vironments with narrower bandwidth, such as distributed computers connected with local area network.

Experiments are performed on the communication pattern found in bitonic sort, that is, when multiple communications of the same distance occur densely, as in Fig. 3. The communication process is simulated on a PC cluster. The cluster consists of 64 nodes, each with dual CPU of Xeon 2.4GHz. The nodes are connected with Gbit-Ethernet. The *phoenix* system [9] was used as the communication library. Communication time was measured changing the number of nodes, with the same total data size. In this situation, the access complexity model predicts linear speedup with growth of the number of nodes. Figure 9 shows the result. The four sequences shown in Fig. 9 represent four different data sizes. The horizontal axis means the number of nodes, and the vertical axis means the ratio between the measured and predicted communication time.

As we can see in Fig. 9, difference of the data size do not seem to affect the results, but the speedup unexpectedly falls down as the number of nodes increases. The current model of the communication channel seems to predict the behavior of environments with enough bandwidth, but do not represent situations with narrower bandwidth. Finding a more precise model of the communication channel, that stays to be simple and general, is still an open problem.

4 Conclusion

A new framework for computational complexity, namely, the *access complexity* model, is proposed. This model is based on the idea that the computational cost really lies in the communication, not in the arithmetic and logical operations.

Therefore, this model defines the complexity of an algorithm as the sum of the communication latencies to read and write data in the memory system of a uniform structure. This model aims to represent various environment, from the memory hierarchy in a single computer, to the network topologies of large scale GRID, in a uniform and simple way. Through analysis of bitonic sort, merge sort, and FFT algorithms, we showed that this model is simple enough to analyze parallel algorithms. Comparing the measured and predicted performances of bitonic sort, we showed that this model can figure out the behavior more precisely than the traditional RAM model. With a simplified FFT program, the access complexity model represents the behavior of actual execution well.

When we analyze the merge sort algorithm, however, the theoretical prediction does not exactly match the actual measurements. We may have to analyze using an algorithm that explicitly simulates the cache mechanism. The current communication channel model does not represent the congestion precisely, when the bandwidth is narrow. These are left for future work. We also plan to confirm the ease of analysis and fitness of the model, through analysis of various other algorithms on various other environments.

Acknowledgement

Discussions with Taiichi Yuasa, Kazunori Ueda, Shinichiro Mori, Masahiko Yasugi, Tsuneyasu Komiya, Nobuya Watanabe, and Norio Kato were indispensable throughout this research project, from the basic design of the model to the design of the experiments. Atsushi Maeda's suggestion was helpful in presenting the measurement results in a more translucent manner.

This research is partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas, 13224050.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.E.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Aggarwal, A., Alpern, B., Chandra, A., Snir, M.: A model for hierarchical memory. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing. (1987) 305–314
3. Alpern, B., Carter, L., Feig, E., Selker, T.: The uniform memory hierarchy model of computation. Algorithmica **12** (1994) 72–109
4. Culler, D.E., Karp, R.M., Patterson, D.A., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: Principles Practice of Parallel Programming. (1993) 1–12
5. Dehne, F.K.H.A., Fabri, A., Rau-Chaplin, A.: Scalable parallel computational geometry for coarse grained multicomputers. International Journal of Computational Geometry and Applications **6** (1996) 379–400
6. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM archive **33** (1990) 103–111

7. Williams, T.L., Parsons, R.J.: The heterogeneous bulk synchronous parallel model. Lecture Notes in Computer Science **1800** (2000) 102–108
8. Rauber, T., Rünger, G.: Program-based locality measures for scientific computing. In: International Parallel and Distributed Processing Symposium. (2003) 164
9. Taura, K., Endo, T., Kaneda, K., Yonezawa, A.: Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). (2003)