# Access Complexity: A New Framework for Computational Complexity

Daisaku Yokoyama†, Nobuya Watanabe‡, and Takashi Chikayama†

†School of Frontier Sciences, the University of Tokyo, Tokyo, Japan
{yokoyama, chikayama}@logos.k.u-tokyo.ac.jp
‡Faculty of Engineering, Okayama University, Okayama, Japan
nobuya@giga.it.okayama-u.ac.jp

***Abstract* — Computational complexity theories have been playing central roles in all areas of computer science. Traditionally, computational complexity is based on the random access memory (RAM) model, in which unit amount of data at an arbitrary location in the memory can be accessed with some fixed constant cost. Recent advances in information technologies made this assumption unrealistic. The speed gap between the processing units and the main memory has been widening dramatically, demanding for deep memory hierarchies. Recent parallel processing systems have more processors than that can cost-effectively share the same memory without cache mechanism. Distributed computation is getting more and more popular. All these demand for a computational complexity model that is more aware of *locality*. In this paper, we propose a new framework for computational complexity, named *access complexity*, in which the cost is in the data transfer than in computation itself. The model is designed so that it naturally reflects the mechanisms used in modern information systems: hierarchical memory, parallel processing, and distributed processing over computer networks.**

***Keywords* — computational complexity, memory model, parallel processing, distributed processing.**

## I. INTRODUCTION

The theories of computational complexity have been playing central roles in selecting one among different algorithms for the same task. If there are two or more algorithms to achieve the same task, an algorithm with smaller computational complexity is preferred because a small difference in the computational complexities of two algorithms makes a large difference in time required to achieve the task as the size of the processed data increases. As algorithm selection is a key issue in all the information technology area, whether the computational complexity theory can actually tell the differences of computation costs is a crucial question.

Traditionally, computational complexity has been aware mainly of the cost of computation. Memory access cost is often estimated based on the random access memory (RAM) model. In the RAM model, data of unit size at any location in the memory can be accessed with some fixed constant cost. As any memory access is for some computation (such as arithmetical and logical operations or making decisions on equality or inequality), if you count cost for computation, you are free to ignore memory access cost as it is counted as a part of the computation cost.

Recent advances in information technologies, however, made this random access assumption unrealistic. The speed gap between the processing units and the main memory has been widening dramatically, demanding for deeper memory hierarchies. Whether the processed data fit in the cache or not makes quite large differences. The difference is often greater than the difference between $O(\log n)$ and $O(\log \log n)$.[1] An auxiliary notion to compensate this problem of the RAM model, called "working set", was proposed and used widely for more realistic performance analysis, but it remains to be a *patch* to the base theory, and cannot be incorporated smoothly with other parts of the theory.

As the clock speed does not increase in the same pace as the circuit scale, parallel processing is getting more cost effective. Parallel processing systems use more and more processors, and thus processors cannot simply share the same memory system smoothly. Memory systems of most of the modern parallel processing systems have non-flat organization; accessing some parts of the memory from one processor is much less costly than other parts of the memory.[2] Software tuning has to make this difference into consideration.

Distributed computation is getting more and more popular. Recent advances in computational grid technologies enable us to use computer systems located quite remotely just like systems in the same computer room. Although this makes software development and resource management much easier, communication cost

[1] Performance is sometimes different at more than an order of magnitude between when all the accessed data fits in the cache and when they are much larger than the cache capacity. Note that, to make $\log n$ ten times larger than $\log \log n$, $n$ has to be greater than $1.79 \times 10^{309}$.

[2] Here, we are talking about physical characteristics of the memory system. The difference between NUMA (non-uniform memory access) and CC (cache coherent)-NUMA architecture lies in the difference in programming ease and not in their performance.

due to the physical remoteness has to be taken into consideration for efficient computation.

All these demand for awareness on *locality*. Accessing memory closely located is less costly than to access memory remotely located. To make the computational complexity useful in such modern situation, its model should be aware of the notion of locality. But which model should replace the good-old, simple and beautiful RAM model? Simply adding the memory access cost to the traditional complexity model would complicate it and may make algorithm analyses much more difficult.

In this paper, we propose a new framework for computational complexity that focuses *only* on data transfer cost. Reversing the traditional cost model, we will *neglect* the cost of arithmetic and logic operations. Computation is costly only because its operands have to be fetched and its result be stored. The cost lies in data access and not in arithmetics.

The model is designed so that it naturally reflects the mechanisms used in modern information processing systems: hierarchical memory, parallel processing, and distributed processing over computer networks.

## II. COMPUTATION MODEL

As stated in the previous section, we consider the costs of data access only, not that of arithmetic/logic operations. This naturally leads us to the following computation model.

### A. *Infinitely Many Processors with Infinite Performance*

As we only care data access costs and arithmetic/logic operation costs are simply ignored, we can assume that there are arbitrarily many processors with infinite processing speed. Processing capacity is only limited by data transfer capacity and not by capacity of operations on them, such as the number of sum-product operations in unit time.

This does not lead to infinitely fast processing. Any operation requires fetching of its operands and storing of its result. These operations require data transfer and thus with some costs.

With this assumption, out model naturally include parallel processing.

### B. *Data Transfer Cost as a Function of Distance*

As we would like to reflect the notion of *locality*, the notion of *distance* has to be introduced into the model. We are yet to know which function represents the real-world systems most appropriately, but the cost should be expressed as a monotonic function of distance.

We decided to start with the function $\log d$ in which $d$ is the distance between two ends of data transfer.

### C. *Memory Space with Finite Density*

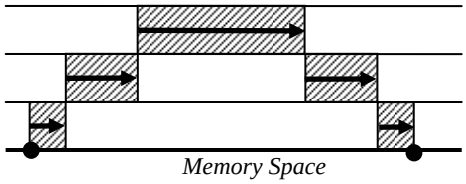


Fig. 1. Layered communication channels: A data transfer activity is initiated at the lowest level, goes to higher levels, and finally goes down again before reaching the destination.

The memory space should have finite density. If its density were infinite, all the data can be stored at a single point and the transfer cost function would be of no use.

What topology the memory space should have is another question. We will start with the simplest of all: one dimensional Euclidean space, that is, a linear space. Data are distributed on this line with some finite density. Computation is performed at somewhere on this line. Any two data items have a non-zero distance between them, and thus computation using them requires some transfer cost.

### D. *Continuity Assumption*

Data storage of actual information system consists of discrete components. For example, first-level cache, second-level cache, and the main memory are discrete components with finite capacities. In the real world, there exists apparent discontinuity between them; performance suddenly goes down when a program starts to access data larger than the cache capacity, for example.

Although this is the reality, taking this discontinuity into account brings a troublesome issue into algorithm analyses. We decided to ignore this and make the memory space model completely planar. There is no singular points on this plane.

### E. *Layered Communication Channels*

Accessing the same or closely located data by many processors should result in congestion of communication. In actual parallel processing systems, this is one of the typical sources of unexpected performance degradation.

The communication mechanism of our model, however, should represent all the different kinds of physical communication channels: wide area networks, local area networks, I/O buses, memory buses, on-chip buses for cache memory access, and even register transfer paths. Congestion in the wide area network should not be an obstacle to register transfer.

We thus decided to give the model multiple layers of communication. Lower layers are used only for local communication. Higher layers are for communication over larger distances. Multiple communication activities can coexist as far as they do not overlap.

Communication activities are initiated at a lowest level. When data should transfer data remotely, the activity

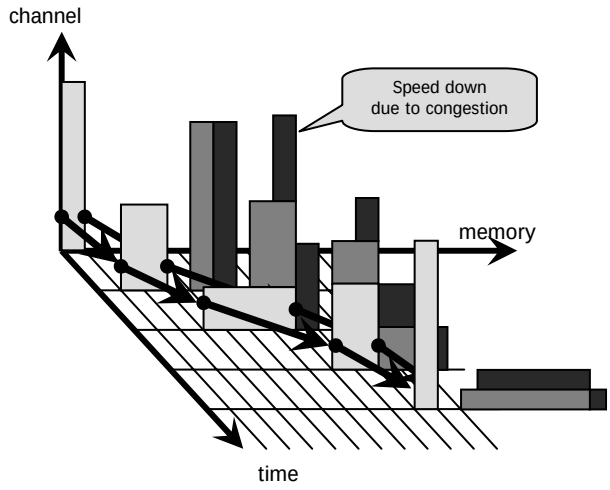


Fig. 2. Summed-up communication channels: A data transfer activity is initiated as a packet with narrow but high effect extent. Its effect extent gradually becomes wider but lower, and then returns to a narrow and low profile before reaching the destination. Loads of different packets are summed up.

would go up to a one higher level. This escalation is repeated until a certain level is reached, and then the activity gradually goes down to lower levels, until it reaches the destination point at the lowest level.

Only those communication activities in the same layer should interfere with each other. For example, register transfer should not slow down network transfer between two other neighboring processors. The width of the area that a communication activity interferes with another should also depend on how remote the data transfer is. Register transfer in one processor does not interfere with that of the neighboring processor, but network transfer in the same segment does.

For simplicity, we assume that all the layers have the same throughput at any given point. However, as interference effects wider area in higher level, total throughput of a layer is higher in lower levels than higher levels. This depicts the fact that, although optical fiber communication has capacity comparable to that of register transfer, there can be millions of processors transferring data between registers within the range of one optical fiber segment.

### *F. Communication Load Sum-Up Model*

The layered communication channel model aims to capture the motion of traveling data packets. It simulates a certain amount of realistic behavior. However, it is still difficult for analyzing some typical algorithms such as FFT, sorting, et al. For this reason, we have designed an even more simplified communication channel model: *load sum-up model.*

Data are transferred in packets, as in previously introduced layered communication channel model. Packets speed up and down exponentially proportional to the time of their travel as in the layered channel model. Thus, communication delay of each packet on vacant channel remains the same. The difference is that the communication channel has only one layer.

In this *load sum-up model*, a packet *consumes up* a part of the limited throughput of the single layer channel, rather than completely occupying a single layer. Each packet burdens a certain area of the channel. The total of the load a packet burdens the channel is proportional to the amount of data in the packet, but the size of the area in the channel that a packet affects is proportional to the speed of the packet at the time; faster a packet, wider area it affects. If the load exceeds the throughput of the channel at any point of the affected area, the transfer of all the packets in the area slowed down. If the load consumes two times of the throughput, communication delay becomes two times longer than ordinary delay.

This model neglects the layered structure of the communication channel of the real world. Thus, a data packet of a local area network interferes with the communication between registers to a certain degree. However, this model very much simplifies the communication behavior and makes communication analysis much easier for some communication patterns.

We propose these two communication channel models, and try to examine the validness and the expressiveness.

## III. ABSTRACT MACHINE

An abstract machine that has the features of the above-described model has been designed. Its specification is described here briefly.

### *A. Memory*

Memory is infinitely large one-dimensional array of *words.* The reason that we did not make it truly continuous nor an array of bits is simply to make emulation of the abstract machine easier and more efficient. The word size can be chosen arbitrarily, but we decided to use 64 bit words, as double-precision floating point arithmetics seems to enable us writing various appropriate benchmark examples.

### *B. Infinitely Many Processors*

As stated above, there are infinitely many processing units distributed densely. Each of such processing units has an ordinary instruction sets, such as arithmetical and logical operations and conditional branches.

To make finite memory density assumption effective while allowing infinitely many densely located processors, processors do not have any local memory, that is, processors do not have registers. Instructions operate only on memory operands and the results are always stored in memory.[3]

[3] To be precise, processors have a program counter. This actually can be used as memory, and can be considered as

### C. Notion of the Current Location

What decide the cost of operations are distances between the location of operation and memory locations to fetch operands from and to store results to. Thus we have to be aware of the location of the operation.

When each instruction is executed, it is executed at a certain "execution location". As the memory consists of discrete *words*, the location is also discrete: it tells at which *word* the instruction is executed.

Each instruction has a field to specify the "next location", which is the place in the data space at which the next instruction is executed.

This should not be confused with the address of the next instruction. The address of the next instruction is normally one next to the current unless explicitly specified in branch instructions, exactly as with conventional processors. The "next location" specifies the location in the data space rather in the instruction space.

### D. Addressing Modes

Operand addressing modes are one of the following.

- Absolute: Absolute address.
- Relative: Relative to the current location.
- Indirect: Indirect addressing on either absolute or relative memory location.

Note that there are no base or index registers as processors do not have any registers.

### E. Instructions for Parallel Processing

Several instructions are added for creation and termination of new threads, and for synchronization between two threads. For synchronization, we incorporated conventional test-and-set to the instruction set.

### F. Communication Costs

Operand fetch consists of two phases. First, the operand fetch request is sent as a packet to the location of the operand. When it is reached at the location, then a packet to actually transfer the requested data is sent the reverse way. For storing the result, only a one-way message is needed.

Any data transfer, including the data fetch request, has communication delay. The cost consists of the minimum delay of one clock for initiating the transfer. Data transfer packet starts from the lowest level in the communication layer, and gradually goes up to higher layers. The lowest level of the communication layers to transfer data in one clock only to the location of the adjacent word. The second layer is twice as fast: it transfers data two words away. The third layer is again twice as fast so that it transfers four words away in one clock, and so on. In general, the $n$'th layer can transfer data $2^n$ words away in a single clock. It should be easy to see that, in total, this realizes $O(\log d)$ delay for transfer of distance $d$, if congestion doesn't make the delay larger.

### G. Emulator

We built an emulator of the above-described abstract machine for carrying out experiments on describing various algorithms and obtain performance data. The address space of the emulator cannot actually be infinitely large, but a full 64 bit address space, which is large enough for any emulation that terminates in a reasonable amount of time.

The emulator is equipped with an assembler, linkage editor, and a visualization tool.

## IV. A Higher Level Language

Using machine language for describing various algorithms is not an easy task. Use of a higher level language is desirable. However, as computational complexity is sensitive to data locations in our model, conventionally used languages such as C are inappropriate; in such languages, variables, both local and global, are allocated at some place that the language processors decide automatically. The program has no control over the decision.

We thus designed and implemented a new language similar to language C but with explicit memory allocation feature. The language is called *Cema*, which stands for "C with Explicit Memory Allocation".

### A. Explicit Stack Allocation

To have control over placement of local variables, Cema provides a primitive to allocate the stack area of a procedure invocation at a specified location. The syntax is as follows.

*proc*(…)@*stack*;

Here, *stack*, a stack type variable whose details are described below, specifies the location of the stack for this invocation of *proc*. Local variables and other working storage required for the invocation is automatically allocated on the stack, as with conventional implementation of C language.

A stack type for a procedure *proc* is denoted as follows.

stack_for *proc*

The size of the area required for invocation of *proc* is automatically computed by the compiler, including areas for any procedure calls from inside it. If *proc* has recursive invocations without explicit memory allocation using the syntax described above, required size cannot be computed statically, and thus the compiler generates an error. Recursive calls therefore always require explicit memory allocation.

---

a defect of the abstract machine. We get around this defect simply by not using such a programming style that utilizes the program counter to hold a meaningful amount of data.

### B. Threads and Synchronization

New threads are created with the following syntax.

`fork` *proc*(…)@*stack*;

A new thread is created and *proc* is invoked within the thread. The thread uses the specified stack. The thread terminates when the control is returned from the invocation.

A special type is provided for synchronization semaphore.

### C. Implementation

A stack frame consists of the following slots.

- Continuation: Instruction address to execute after finishing the procedure invocation of this stack frame. This is a generalization of return address.
- Return Location: Execution location for execution after returning from the invocation of this frame. While a procedure is executing, execution location is the top of the stack frame. Thus, this slot corresponds to the stack frame link in conventional implementations of C-like languages.
- Working Area: For local variables and temporary storage for evaluating expressions.

## V. Preliminary Evaluation

### A. Analysis of bitonic sort

In this section, we analyze the computational cost of the bitonic sort algorithm on our model. Bitonic sort is one of the sorting networks. Sorting of $n$ elements costs $O(n\log^2 n)$ in RAM model, and $O((n\log^2 n)/P)$ in PRAM model with $P$ processors.

The layered communication channel model is too complicated to analyze its communication pattern. Thus, we use the load sum-up model here.

Initially, the data to be sorted is placed in a one-dimensional array. Every bitonic merge (n) stage has communications of two data that are placed $n/2$ units apart. These communications have latency of $O(\log n)$. Thus, the whole bitonic sort (n) operation costs $O(n\log^3 n)$.

For analysis of parallel execution, we have to consider communication congestion. Every communication in bitonic merge (n) has the distance of $n/2P$. Each packet has the same speed and the same load distribution. The load of the communication channel becomes highest when we use $n/2$ processors. Even in this situation, the load does not exceed that of when only one packet is transferred, that is, when the operation is sequential. The load does not consume up the whole throughput, at anywhere and anytime. Congestion does not occur at all and thus, bitonic sort (n) costs $O((n\log^3 n)/P)$.

### B. Benchmarks on real machines

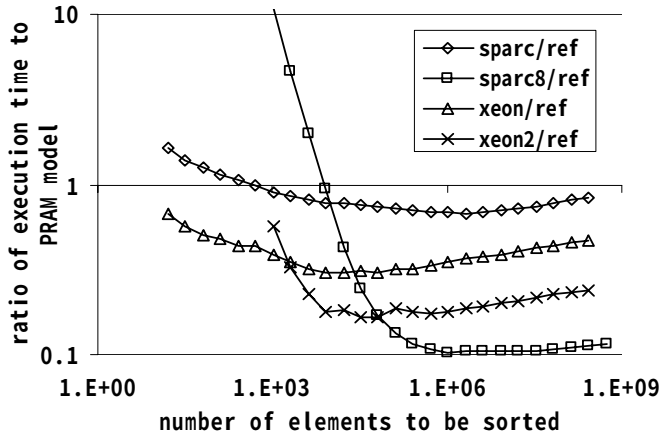


Fig. 3. The bitonic sort algorithm compared to the PRAM model. Sparc means using one CPU, sparc8 means using 8 CPUs, and so on.

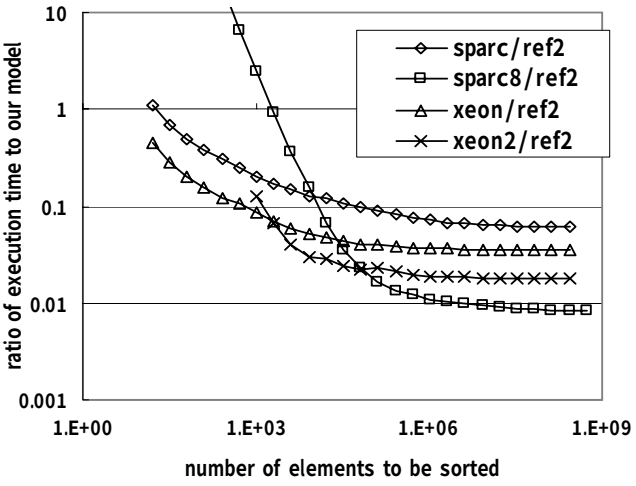


Fig. 4. The bitonic sort algorithm compared to our model.

We implemented the bitonic sort algorithm on some parallel machines, and examined the fitness of the analysis described above. Target systems are:

- Ultra SPARC III Cu, 1.2GHz, 8CPU.
- Pentium 4 Xeon, 2.4GHz, 2CPU.

Both are shared memory machine. The program is written with the *pthread* library.

Figure 3 shows the execution time ratio compared to the PRAM model. The difference of the absolute value of each machine is meaningless. As clearly seen in the figure, the ratio is increasing at large data sizes.

Figure 4 shows the same data compared with our model. The ratio seems to be converging to a constant as the data sizes increase. Our data model to have correctly captured the behaviors of different systems of the real world.

These experiments are all on shared memory machines. These machines have quite powerful communication channels compared to their parallelism. PC clusters, in

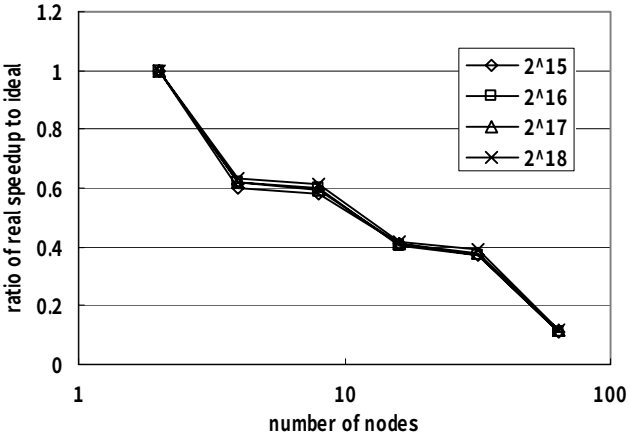


Fig. 5. The ratio of real speedup compared to the ideal.

contrast, do not have such a powerful communication bandwidth.

We tested the communication part of the bitonic sort algorithm on a PC cluster. The program only transfers data in the communication pattern required in the algorithm. The program is written with *phoenix*, a communication library for distributed computation. The test environment is a system consisting of 64 node each with dual Xeon 2.4GHz processors. Nodes are connected with a Gbit-ether network.

The result is shown in figure 5. The ratio of real speedup to ideal speedup is getting smaller. This means overwhelming data packets brought in measurable communication delay. Our model could not represent this behavior.

The decline looks consistent; adjustment required to our model might be simple. The layered communication channel model might also fill the gap. Evaluation of the applicable range of our model is our future work.

## VI. RELATED WORK

In 1987, Aggarwal *et al.* proposed a sequential processing model with non-random access memory model and showed complexity analysis on the model [1]. In their model, access cost of data at address $x$ is proportional to log $x$. They showed an algorithm to realize the same complexity as in the RAM model for matrix multiplication, that is, $O(n^3)$, where $n$ is the size of one dimension of arrays. They also showed that, for problems that requires less repeated accesses to the same data cannot keep the efficiency of algorithms for RAM. For example, FFT and sorting have the complexity of $O(n \log n \log \log n)$ on their model, rather than $O(n \log n)$ achievable on RAM.

Parallel random access machine (PRAM) model has long been investigated by many researchers on parallel algorithm design area. PRAM is a straightforward extension of the RAM model to parallel processing and has the same problem with the RAM model that memory access locality is not taken into consideration in the analysis.

Cullar *et al.* proposed a frame work named LogP, which is a simplified distributed memory model [2]. The model characterizes a parallel processing system by four parameters. Although this simplification makes analysis easier while keeping access cost into consideration, it does not take distances between processors into account, and, more crucially, does not take memory access costs within each processor into account.

In a sense, our model is an extension of Aggarwal's work so that it can analyze parallel algorithms in a unified manner.

### C. Conclusion

A new framework for computational complexity is proposed. The processing costs taken into consideration in the model are only those costs of data communication. Costs of computation are simply ignored.

An abstract machine that fits the model has been given. It is shown that ignoring computation costs does not lead to an unrealistic model.

The design of a higher level language for describing algorithms on the abstract machine is also reported. The language is reasonably easy to describe complicated algorithms while allowing explicit memory allocation, which is a key to efficiency of algorithms on the model.

We are yet to prove the model to be actually useful. The following two issues have to be investigated.

- Ease of Analyses: whether analyses of algorithms under the model is not too hard.
- Fitness: whether the results of analyses on the model actually represents real-world performance well.

To see the above two, more experiments of describing various algorithms and analyzing them on the model are to be done.


## ACKNOWLEDGEMENT

Discussion with Taiichi Yuasa, Kazunori Ueda, Shinichiro Mori, Masahiro Yasugi, Tsuneyasu Komiya, and Norio Kato has been helpful in designing the model and the Cema language.

This research is partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas, 13224050, 2003.


## REFERENCES


[1] A. Aggarwal, *et al.* "Hierarchical memory with block transfer," *Proc. 28th Symp. on Foundations of Computer Science*, 1987.

[2] D. Cullar, *et al.* "LogP: A Practical Model of Parallel Computation," *CACM 39-11*, 1996.