UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

Programming with a Parametrized Parallel Programming Model

Simo Juvaste

Report A-1997-4

ACM D.1.3; D.3.2
UDK 681.3.06
ISSN 0789-7316
ISBN 951-708-512-5

Programming with a Parametrized Parallel Programming Model

Simo Juvaste

juvaste@cs.joensuu.fi
University of Joensuu, Department of Computer Science
PO Box 111, 80101 Joensuu, FINLAND
http://www.cs.joensuu.fi/~sjuva/f-pram/

Abstract: We present a new experimental framework, the F-PRAM model, for modeling and studying the effects of different properties of parallel machines. The goal of the new model is to provide programmer-level facilities for more accurate and portable computation-communication balancing. As a computation model, F-PRAM provides shared memory but guides the usage of the shared memory by a set of parameters. In comparison with the BSP model, we provide more asynchronous execution and model the costs and delays induced by communication bandwidth restrictions, communication latency, overhead, and synchronization independently of each other. We have written a high level language compiler and a parametrized emulator to measure the effects of different properties of parallel machines on different parallel algorithms.

Keywords: F-PRAM, Parallel, Programming, Computation, Model, Compiler, Emulator

1 Introduction

Most of the current research on parallel computing is based either on simplified models of computation, or on some existing physical computer architectures. The simplified models, e.g. the PRAM model, easily guide to design too finegrained algorithms, and thus often result in programs that are not efficient on current parallel computers. On the other hand, the application programming for the existing parallel computers usually results in rather unportable programs. Further, because of the lack of an applicable theoretical algorithmic background, the ad hoc programs rarely result optimally. Since the parallel computing is vital especially for the most demanding applications, we should optimize our algorithms for both the speed and the efficiency of the computations. In addition, we should be able to find the most important and cost-effective features of parallel computers to be able to develop cost-effective parallel computers in the future.

Several suggestions have been made to fill the gap between the theoretical algorithms and the existing parallel computers. Some examples of the proposed models are BSP [9], LogP [3], Y-PRAM [8], APRAM [2], and Phase PRAM [4]. Each of these models provides a set of restrictions and other features which should make the model more realistic. Some of the models are, however, still rather theoretical, and do not necessarily provide good enough connection between the parallel programming and parallel computers. In this paper we shall present a new model of parallel computation, which models the parallel computers more accurately, but is not overly difficult to use. Our main goal is to provide facilities for the algorithm designer to more accurately balance and interleave the computation and the communication in different types of parallel computers. The balancing is done in co-operation by the programmer and the compiler.

Compared to the BSP model, we do not require the constant length supersteps, but instead we let the computation continue asynchronously until an explicit barrier synchronization is called. The length of a BSP superstep, and thus the minimal latency, is large, because during a superstep the computer must be able to perform a full h-relation and synchronize all processors. Latency of our model is only the latency of a single shared memory reference. In addition, we handle the costs and delays induced by communication bandwidth restrictions, communication latency, and synchronization independently of each other. Compared to the LogP model, we provide shared memory and concurrency of computation and communication, both of which are only hinted in the LogP suggestion [5]. The parameters are more or less similar, except that F-PRAM model has an explicit synchronization primitive with a cost instead of the synchronization via message passing. Further, our model includes a set of secondary parameters that can be used for more accurate modeling of parallel computers.

In the rest of this paper we shall overview the computation model in section 2, define the language FPM in section 3, give a larger example in section 4, and finally conclude and sketch the future research in section 5.

2 F-PRAM model of parallel computation

We call our new model *F-PRAM model* since it uses so-called *futures* for shared memory references. They were originally introduced by Halstead [5] as a primitive of parallel computation in the Multilisp language.

The F-PRAM model consists of a computation model and a cost model. The computation model defines the components modeling the components of the parallel computers and the execution of the algorithms on the model. The cost model consists of an algorithm cost model and a machine cost model. In this paper we shall concentrate on the algorithm cost model, that consists of a set of parameters that characterize the properties of the parallel computers. Based on the machine-dependent values of the parameters, the algorithm cost model charges machine-dependent costs on the different operations and algorithms. Consequently, the algorithm cost model tells us which properties affect the execution cost of an algorithm most, and, therefore, what kind of algorithms we should write for different types of parallel computers. Furthermore, the parameters of the cost model allow us to write algorithms that are capable to adapt themselves to different parallel machines. Using the algorithm cost model, we can answer to questions of types "What requirements this particular application sets on the parallel machine to be executed efficiently?", "How efficiently this algorithm can be executed on an Ethernet-connected workstation network?", "Should we buy more processors or a faster interconnection network?", or "What properties of the parallel computers could we weaken without decreasing the performance of our algorithm?".

Basic structure of the new parallel computation model

Components of any parallel computer include a number of processors, an amount of local and/or global memory, an interprocessor communication facility, and some I/O facilities. The F-PRAM model naturally includes these, and actually little else. Figure 1 presents the conceptual structure of the F-PRAM model, as well as the conceptual structure of an instance of the F-PRAM model. In this subsection we shall present the responsibilities of each of the components of the model. Further, we shall define the interactions between the components. We must remember, however, that the presented structure needs not to be the structure of the parallel computers we are modeling of. For example, the shared memory modules can be located within the processing nodes, and the synchronization can be accomplished using the interconnection network.

The computational power of the F-PRAM consists of a limited, machine dependent, number of processing nodes, each of which consists mainly of an ordinary sequential random access machine (RAM). Each RAM has a private memory for the program and local data.

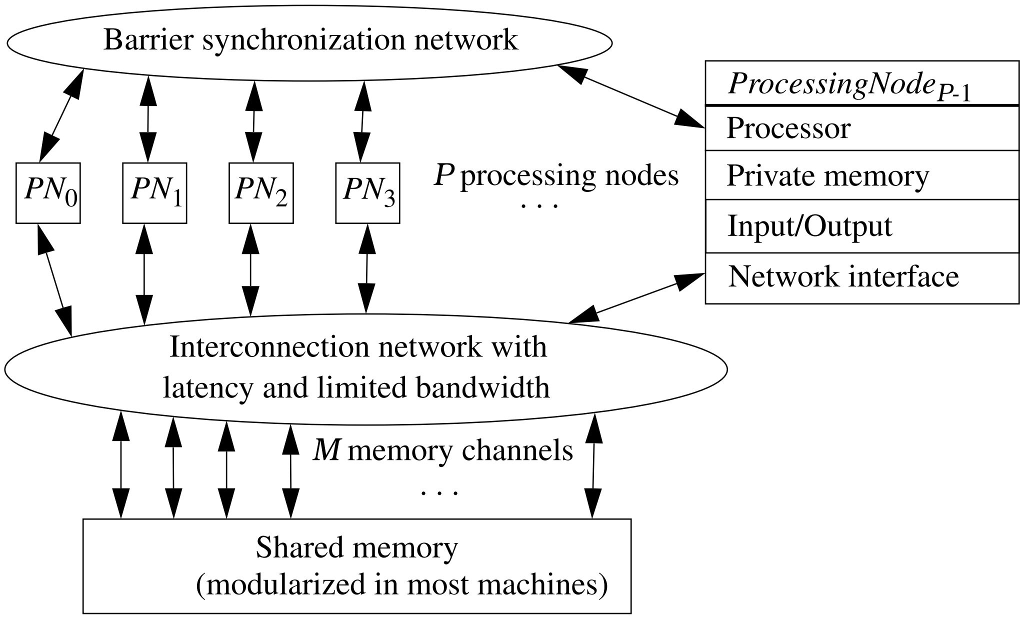

Figure 1: The conceptual structure of the F-PRAM

The exact architecture of the processor is irrelevant with respect to the model since the model does not require any special features, such as special registers, multithreading, fast context switching, or special fast interrupts. This is because each of the processors always works sequentially on only one instruction stream. Also, the processors work independently until they have a need to interact with another part of the machine.

In addition to the normal RAM operations, an F-PRAM processor is able to issue shared memory references, *futures*. The processor delegates the shared memory references to the network interface, which is the only additional component in the processing node compared to a RAM. Whenever a processors needs a value from the shared memory, it requests it in advance using a future. Using a programming language notation, the issuing of a future is written as

future `local_variable := shared_variable;` (1)

which takes constant time, but does not yet assign the value of the `shared_variable` to the `local_variable`. Instead, the `local_variable` is assigned a placeholder, a *future*, which is not a usable value. After the issuing of the future, the processor continues its execution, and it may issue more futures. The interconnection network and the shared memory are responsible for the actual resolving of the future. Within L time units the placeholder should get replaced by the actual value. After the replacement the `local_variable` is just as any ordinary local variable. It is programmer's responsibility to ensure that the processor has enough work to do before it tries to use again the `local_variable`. As we shall see later, the programmer can do this because he or she can use the value of the parameter L in the program. Shared memory writes take only constant processor time, but the actual effect on the shared memory location will take place only after at most L time units.

The shared memory can be either monolithic or modularized. In reality, the true monolithic M-port memory is not feasible currently, thus all existing parallel computers have several memory modules, each of which is connected to the interconnection network. Our current emulator simulates the shared memory and the interconnection network by a queue of references. The rate at which the references are filled is determined by the bandwidth inefficiency and the number of processors.

Even if the shared memory is the apparent communication medium of the processors, the references are delivered by the interconnection network. Different parallel computers use different types of network topologies, thus we shall not make any assumptions on the actual topology. Instead, we shall model the capabilities of different interconnection networks. The most important properties of an interconnection network are the bisection bandwidth, the capacity, and the latency of the network. Since we are interested in the ability of the network to deliver global messages between the processing nodes and the shared memory, we combine the first two measures. Further, since the available total bandwidth should scale up with the number of processors, we shall use the shared memory bandwidth available for each processor. With respect to the latency, we are interested in the time after which all issued references are delivered with high probability.

Since we use shared memory references for interprocessor communication, the references must be synchronized. Consequently, an important part of the interprocessor communication is the synchronization network. We shall model the synchronization network by the time it takes to barrier synchronize all processors.

Algorithm cost model

The above definitions characterized the qualitative properties of the F-PRAM model. In this subsection we shall define the algorithm cost model, that models the quantitative properties of the F-PRAM and the parallel machines we are modeling. The algorithm cost model consists of a rather large set of parameters, each of which models a property of the parallel computer. The parameters tell us the values of the properties in a particular machine, and thus the costs that are charged for different operations when modeling the parallel machine. The parametrized algorithm cost models, such as the F-PRAM, yield more accurate estimations on the execution cost of an algorithm in a given machine than the uniform-cost models. Besides, since the parameters are available for the algorithm designer and for the algorithm, we can design portable algorithms that optimize themselves according to the properties of the machine they are executed by.

By default we suppose that the parameters are machine-dependent constants, but as an interesting option we could think of allowing them to change dynamically during the execution of the algorithm. This way the algorithms could adapt to changes in the load of the parallel computer.

To ease up the design and the analysis of algorithms, we shall divide the parameters to primary and secondary parameters according to their importance. Table 1 presents the parameters of the parameters of the F-PRAM model and gives some hints on their typical values.

Primary parameters model the most important properties of the parallel computers and have to be considered always when designing or analyzing an algorithm or a program. Moreover, the primary parameters do not have any default values since they are very machine-dependent. The parameter L, the *latency of shared memory references*, stands for the expected worst time needed for a shared memory reference to be completed. The parameter B, the *bandwidth inefficiency*, stands for the inverse of the maximum shared memory referencing capacity available for each processor, in operations/word. The parameter S, the *synchronization delay*, stands for the time needed to barrier synchronize all processors.

Table 1: The parameters of the F-PRAM model

	Primary parameters	Typical values
P	The number of processors	
L	The latency of shared memory references	$\geq 2 \times \varnothing$ *of interconnection network*
B	The bandwidth inefficiency	≥ 1, e.g. $P/M \times Speed_P/Speed_M$
S	The synchronization delay	$\geq L$, even $L \times \log P$
	Secondary parameters	
B_V	The single variable bandwidth	≤ 1
B_M	The single memory module bandwidth	≤ 1
M	The number of shared memory modules	P
B_{IO}	The input/output bandwidth inefficiency	≥ 1
B_P	The processor communication overhead	≥ 1
B_B	The block reference bandwidth inefficiency	$< B$

The role of the *secondary parameters* is to enable the possibility for more refined analysis of our algorithms on parallel computers. Also, the secondary parameters allow us to examine the effect of some strong features of special parallel computer architectures to see how much they would speed up our algorithms. To allow future improvements on the model, we do not yet bind the set of the secondary parameters tightly. To maintain reasonable portability and analyzability, however, we should design the basic algorithms without extensive use of the secondary parameters. Formally, the design without the secondary parameters corresponds to use of the secondary parameters with their default values. Accurate definitions can be found in [6]. The *block reference bandwidth inefficiency* models the possible improved throughput of futures longer than one word.

3 Programming language FPM for F-PRAM

To test the usability of the F-PRAM model, we defined and implemented an experimental programming language and a compiler. The programs written in FPM are compiled to F-PRAM machine language for execution. Currently we use an emulator [10] to execute the F-PRAM machine language.

The programming language FPM is a superset of a subset of Wirth's Modula-2 [11]. Main differences to Modula-2 are parallel loops and separate shared and private variables. Because FPM is intended for F-PRAM programming, programmer has to think a bit in terms of F-PRAM when writing a FPM program. It is left partly to programmer's responsibility to use all processors correctly. Full definition of the language with examples can be found in [7].

Parallelism extensions to Modula-2

A normal FPM program has a sequential body and one or more parallel sections within the sequential body. In other words, the execution proceeds sequentially until the thread reaches a `par-do` statement, the whose body is executed in parallel, and proceeds again sequentially after the `par-do` statement. Actually, however, all processors execute the same code, i.e. the whole program, and the `par-do` statements only control and reindex the processors.

Par-do statement

A `par-do` statement equals to the `for-do` iteration statement except that if the execution thread has more than one processors available, the iterations are executed as parallel as possible. If the number of available processors is at least as big as the range of the `par-do` statement, all iterations are executed in parallel. Otherwise one or more processors take care of two or more iterations within the range of the `par-do` statement.

The `par-do` statements may be nested arbitrarily or even recursively. Each `par-do` statement distributes the available processors evenly among the values in the iteration range to execute the body of the `par-do` statement. On the outermost `par-do` statement there are P processors (see below) available. On the inner `par-do` statements there are less processors available, depending on the sizes of the ranges of the outer `par-do` statements. The execution time of a `par-do` statement is $c+T\times n/P$, where T is the time used by the body of the `par-do` statement, n is the size of the range, P is the number of available processors, and c is the start-up time of the `par-do` statements. Using the current FPM compiler, the start-up time of a `par-do` is 30-80 clock cycles depending on the optimization level, the number of available registers, and the number of available processors compared the range of the `par-do` statement. The time is required for each processor to calculate the new number of processors, processor-ids, the values of the iteration variable it has to go through, and to save all the information to the `par-do` stack.

Since the processors are able to start the `par-do` statement independently, the processors are not required to enter the `par-do` statement synchronously. Consequently, a processor may even exit the `par-do` statement before another processor enters it.

Synchronization

In most parallel programs it is vital that some operations are executed in a certain order. Because processors work asynchronously and communicate through the shared memory, one processor can, for example, read a memory cell before an other processor has written to it. As processors communicate only through the shared memory, we must make sure that all processors execute the vital statements simultaneously. The FPM language exploits the barrier synchronization property of the F-PRAM for synchronization purposes.

The statement `synchronize` causes processors to wait until all processors have entered it, and after that every processor exits it simultaneously. The time required by the synchronization depends on the value of the parameter S of the F-PRAM used to execute the program. Synchronization has to be placed in such a place where all processors surely enter. If a processor skips synchronization all others will wait for it forever.

Shared variable references

The shared memory of the F-PRAM model is not directly accessible to the processors. Instead, the processors may request values from the shared memory by the future instruction and write to the memory by the fwrite instruction. Similarly, the FPM language supports `future` statement for requesting values from shared variables to the local variables and `fwrite` statement for writing values of local expressions to the shared variables.

A `future` statement requests the interprocessor network to deliver the value of the shared variable to the local variable. The variables may also be members of structured variables or arrays. The processor is not occupied during the delivery. Instead, the processor may do other useful work during the L clock cycles the delivery takes. During the delivery the local variable is unreadable. If the processor tries to use the local variable before the value is delivered, it stops to wait. The readiness of a future can be checked using the function `fcheck()`, which returns true or false depending whether the local variable given as the parameter contains a readable value or an unreadable future-pending value.

A `fwrite` statement evaluates the local expression and sets a write request to the shared memory on the way. The processor is occupied only unit time, but the write takes effect in the shared memory only after at most L clock cycles.

Predeclared variables

Each of the parameters of the F-PRAM model has a corresponding predeclared read-only variable in the FPM language. The values of the variables are assigned by the F-PRAM machine at start-up time, and they represent the actual properties of the machine. Currently implemented parameter variables are `P`, `B`, `L`, `S`, `BP`, and `BV`. See table 1 for the definitions of the parameters. The variable `P` stands for the number of available processors. In other words the number of processors execution the current thread, and available for the parallelization of the next `par-do` statement. The number of processors on previous levels of `par-do` statements is available using the function-like expression `P(level)`, where `level` stands for the depth of `par-do` statements, `P(0)` being the outer level outside any `par-do` statements, i.e. the number of processors in the whole machine. Variable `LBL` (Loop Body Length, a better name yet to be found) can be used to more accurately balance the communication and computation. The value of the variable approximates the number of clock cycles required by the body of the innermost `for-do` loop. The typical use of the variables would be to use `for-do` iteration of length `L/LBL` as a pipeline to adaptively hide the latency.

4 Example: parallel mergesort

As an example we use the classic parallel sorting, odd-even mergesort algorithm by Batcher [1]. The basic algorithm sorts N elements in $O(\log^2 N)$ steps using N processors. If we have less than N processors, we terminate the `par-do` recursion and sort and merge sequentially as soon as we have run out of processors. This is more realistic with current parallel computers and problem sizes. Moreover, the basic odd-even mergesort uses a factor of $O(\log P)$ more work than the sequential one. Using less processors, we achieve better efficiency.

Using the procedure Odd-even_merge of the algorithm 1 we can merge N elements with P processors in time

$$T_{oemerge}(N, P) = \begin{cases} T_{oemerge}(N/2, P/2) + N/P, \text{ if } P > 1, \\ N, \text{ if } P = 1 \end{cases} \tag{2}$$

$$= (\log P + 1)N/P \in O(N \log P/P) . \tag{3}$$

Thus we can sort N elements with P processors in time

$$T_{oemsort}(N, P) = \begin{cases} T_{oemsort}(N/2, P/2) + T_{oemerge}(N, P), \text{ if } P > 1, \\ N \log N, \text{ if } P = 1 \end{cases} \tag{4}$$

$$= \log P \times (\log P + 1) \times N/P + (N/P) \times \log(N/P) \tag{5}$$

$$\in O((N/P) \times (\log^2 P + \log N/P)) \tag{6}$$

```
procedure Odd-even_mergesort (A : array[1..N])
    if Processors = 1 then
        Sequential_mergesort(A)
    else
        par i = 1 to 2 do
            Odd-even_mergesort(i:th half of A)
        Odd-even_merge(halves of A)

procedure Odd-even_merge ( A : array[1..N])
    if Processors = 1 then
        Sequential_merge(A)
    else
        par i = 1 to 2 do
            Odd-even_merge(halves of odd(A) & halves of even(A))
        par i = 1 to N-1 by 2 do
            pipelined_compare-exchange (A[i], A[i+1])
```

Algorithm 1: The modified algorithm of parallel odd-even mergesort

assuming that B=1 and L=1, i.e. using the PRAM-like shared memory performance. The actual program written in language FPM can be found in [7]. Figure 2 presents the practical performance of the program on different sizes of input and on different number of processors. The first graph presents the relation of the measured execution time and the asymptotic execution time given in formula (5). In practice, we can see that a step (comparison, exchange, etc.) of the algorithm requires 30-60 clock cycles on the emulator on the actual program, unless we use nearly as many processors as we have elements to be sorted. When using a relatively high number of processors compared to the input size, the initialization time of the `par-do` statements dominates the whole execution time, thus resulting peaks in the graph. The next graph presents the work per element to be sorted for different problem and machine sizes. The last graph shows the speedup compared to the number of processors and theoretical speed-up of the asymptotic execution time derived from formula (5).

Effect of bandwidth inefficiency B and latency L

The above analysis on the execution time on different number of processors contains no new information on parallelism. Instead, the purpose of our model, emulator, and compiler is to study the effect of other properties of the parallel machine on the execution. Here we shall present the effect of the shared memory bandwidth inefficiency and latency. Analyzing the shared memory usage of the algorithm 1, we can derive a new formula for the execution time. The new formula will include parameters B and L. The analysis is naturally a bit more difficult, and since the shared memory references occur concurrently with the computation, the formulas include maxima over the parameters and the time required by the computation. Taking the time required by the computation into account induces also some constants to the formulas.

$$T_{seq\_merge}(N) = \max(L, N) + N\times\max(C_1, B) + \mathrm{L} \qquad (7)$$

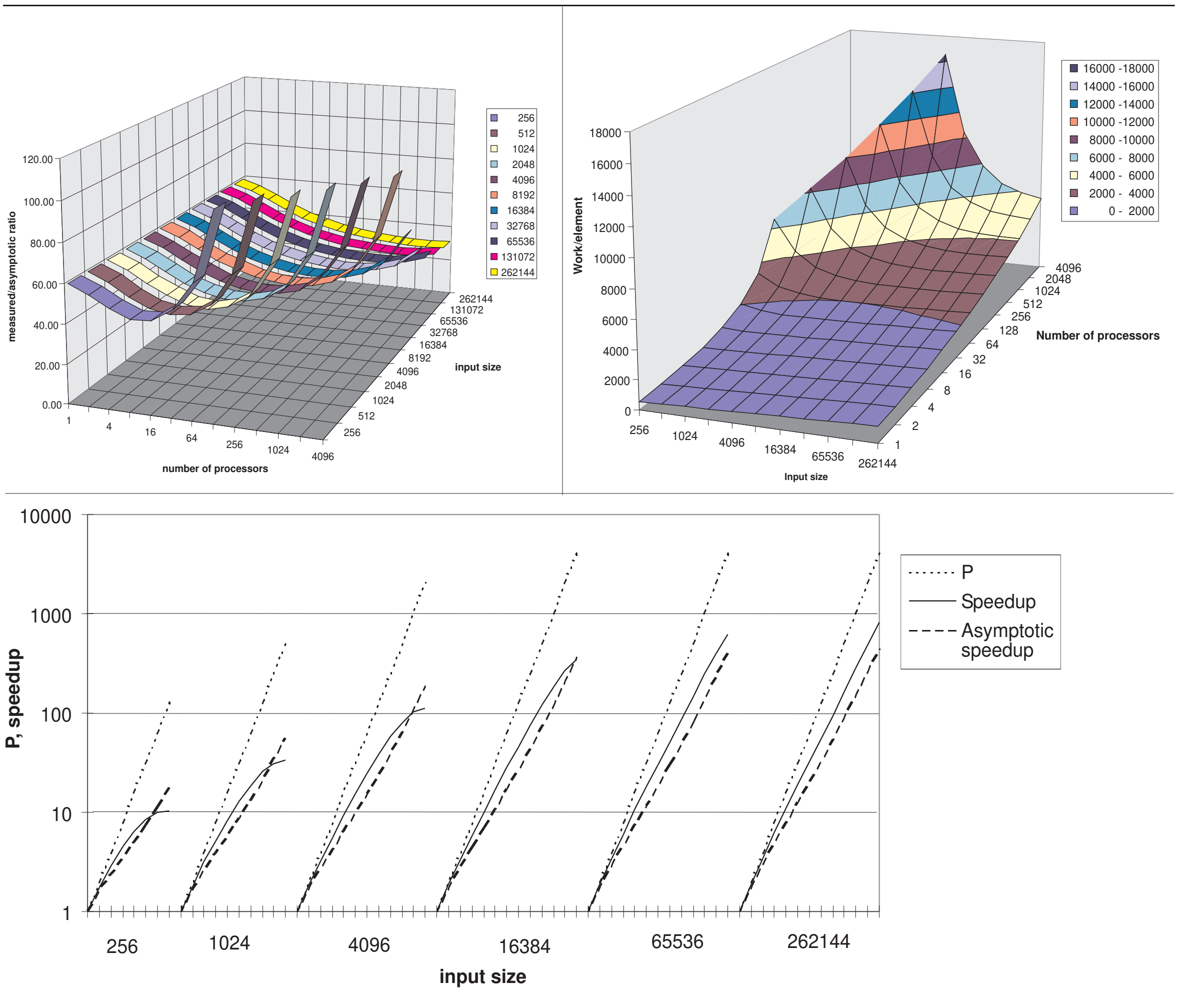

Figure 2: Performance of the algorithm in an ideal machine with B=1 and L=1

$$T_{seq\_msort}(N) = \max(L, N) + N\log N + \max(L, NC_2, NB) \tag{8}$$

$$T_{oemerge}(N, P) = \begin{cases} T_{oemerge}(N/2, P/2) + 2\times\max(L, NC_2/P, NB/P), \text{ if } P > 1, \\ \max(L, N) + N\times\max(C_1, B) + \text{L}, \text{ if } P = 1 \end{cases} \tag{9}$$

$$\leq 2(\log P + 1)\max(L, NC_2/P, NB/P) \tag{10}$$

$$\in O((L + BN/P)\times\log P\,. \tag{11}$$

Using the above components, we can sort N elements with P processors in time

$$T_{oemsort}(N, P) = \begin{cases} T_{oemsort}(N/2, P/2) + T_{oemerge}(N, P), \text{ if } P > 1, \\ T_{seq\_msort}(N), \text{ if } P = 1 \end{cases} \tag{12}$$

$$\leq \log P\times 2(\log P + 1)\times\max(L, NC_2/P, NB/P) + \max(L, N/P) + N/P\times\log N/P + \max(L, NC_2/P, NB/P) \tag{13}$$

$$\leq 2\times(\log P + 1)^2\times\max(L, NC_2/P, NB/P) + N/P\times\log N/P \tag{14}$$

$$\in O((L + BN/P)\times\log^2 P + N/P\times\log N/P) \tag{15}$$

Since we cannot derive the constants involved in the above formulas, the formulas are actually of pretty limited use. We can see from formula (15) that when using large values of P and L or B, the term $(L + BN/P)\times\log^2 P$ begins to dominate, thus making the execution inefficient. The formulas using the ordo notation destroy some important information on the formulas, namely the fact that up to some limit, the values parameters L and B do not affect much on the performance of the algorithm. Unless we have knowledge on the constants, we cannot predict the limits of usable values of L and B. Using the compiler and the emulator, we can measure the actual performance of the algorithm using different sets of parameters. Figure 3 presents the measured effect of the parameters B and L for the same problem size (16,386) and for different number of processors (16, 64, 256, and 1024). On each graph, the

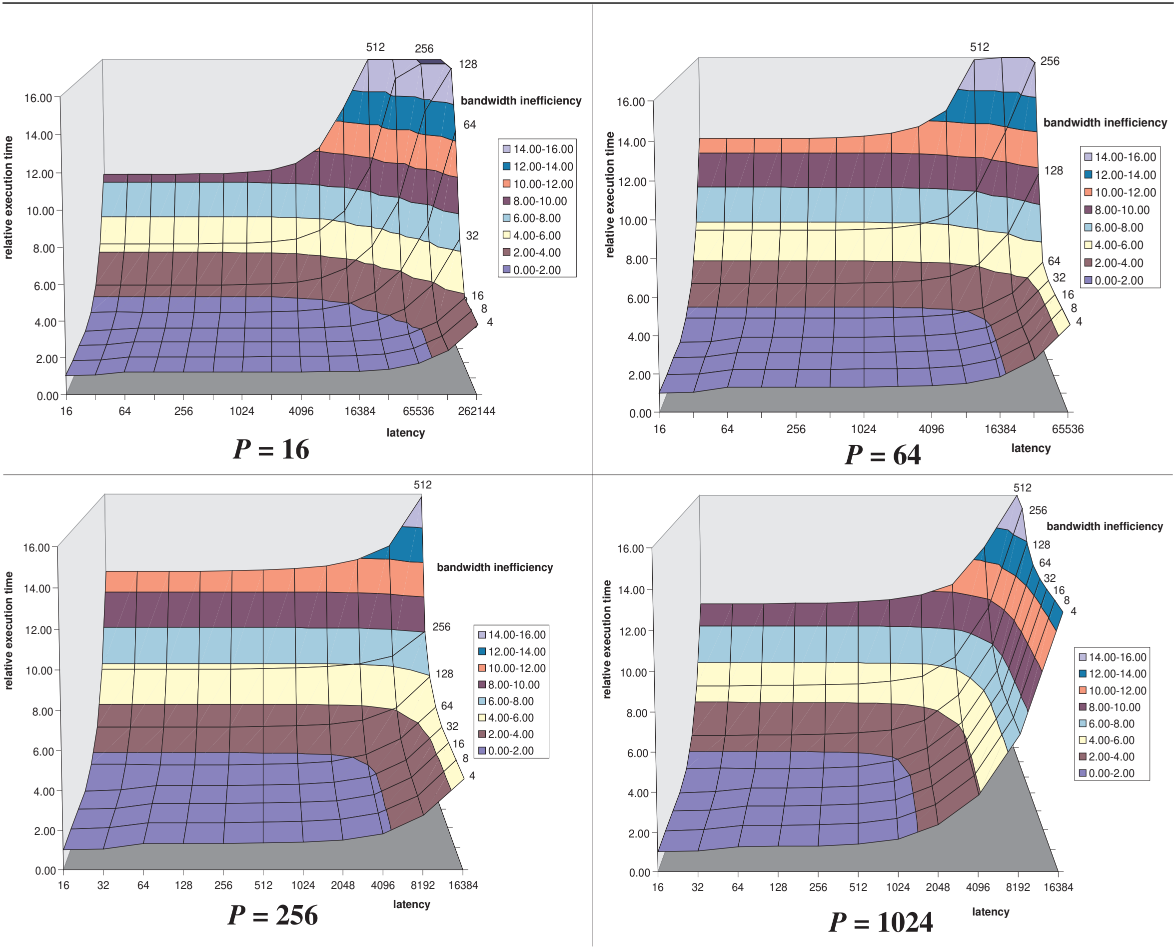

Figure 3: Combined effect of B and L for different number of processors

execution times are proportional to the time used by case B=1 and L=1. The lower plateaus appearing in mid-gray present the sets of parameters which at most double the execution time. We can see that as P increases, the effect of L increases, i.e. the more processors we use, the less latency the algorithm tolerates, as we can see also from formulas (14) and (15). On the other hand, as P increases, the effect of B does not increase, which is due the fact that the parameter B presents the bandwidth inefficiency per processor. Consequently, increasing the number of processors while keeping B constant, increases the total shared memory bandwidth of the whole machine. As a result of this, the effect of B is more dramatic on the machines with fewer processors.

Effect of processor communication overhead B_P

We shall present the parameter B_P, processor communication overhead, as an example of the effect of the secondary parameters. Considering the importance of the overhead, the

LogP model includes it, whereas BSP ignores it. If the processing nodes of a parallel computer do not have dedicated hardware for the interface to the interconnection network, the processor must inject the shared memory references to the interconnection network itself. This will take some time as the processor possibly have to change context for the interconnection handler, negotiate with the channels, inject the message, and return to the main execution thread. This takes easily dozens of clock cycles, and this will happen on every shared memory reference. Considering the algorithm 1, the parameter B_P is multiplicative on every stage except on sequential sorting. We can show that the asymptotic execution time is

$$O((N/P)\log^2 P \times B_P + N/P \times \log N/P)\ , \tag{16}$$

assuming that B=1 and L=1. In other words, even if we were using only a few processors, big values of overhead B_P make the execution inefficient. As with the previous formulas, we cannot predict the allowed constants of the formula (16). Instead, we have measured the actual effect using the emulator. Figure 4 presents the measured effect for different numbers of processors for input size 16,386. We can see that unless we use very few processors, overheads as low as 32 have a significant effect on the efficiency of the sorting. Comparing overhead B_P to the latency L, the algorithm can tolerate two orders of magnitude larger latency than overhead. Consequently, it is vital that the shared memory references are executed concurrently with the computation.

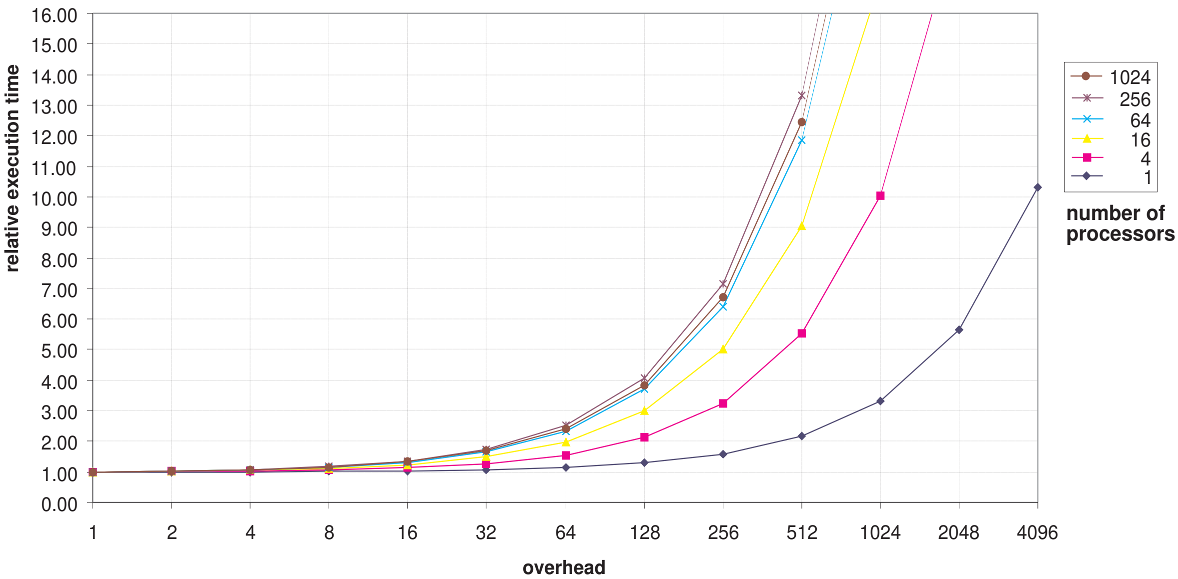

Figure 4: Effect of overhead B_P on execution time, sorting, N=16,386

As a conclusion of the measurements, we can see that all of the parameters can have a dominating effect on the execution time if they are allowed to grow enough. Above some limit, the effect of the parameters is practically linear, i.e. multiplicative. The limits are dependent on the coarseness of the computation, and on the other parameters. To some extend we can derive these limits, or at least some properties of the limits, analyzing the algorithms using some constants. For more accurate results we must measure the actual effects.

5 Conclusions and future research

We have defined a new model to support balancing and interleaving of computation and communication in parallel computers. The advantage of the F-PRAM model is that in addition to model of computation, the model spans from the algorithm costs model to the programming model and to the modeling of parallel computers. Even if the model provides an accurate set of cost parameters, it is not much harder to program than the unit-cost models.

To support the model, we have also defined a programming language for writing working and compileable programs out of our algorithms. We also have a compiler and an emulator to verify and especially benchmark our algorithms. The measurements of the algorithms are important since the ordo analysis hide some of the important information of the effect of different machine properties, and the analysis with accurate constants may be difficult.

The futures, i.e. concurrent computation and communication, allow us to write algorithms that tolerate much larger latencies than if the processors had to wait idle during the shared memory reference. This is analogous to the asynchronous message-passing, but using futures we can program using the more familiar shared memory model.

Our future work is to analyze and use the model more extensively to give it some more credibility. The model itself will probably remain mostly unchanged. In addition to the basic algorithm complexity analysis with the unfixed parameters, we shall also experiment on actual parallel machines with their F-PRAM parameters. To improve practical performance on current parallel computers we should especially exploit the parameters B, B_B, B_M and M.

An interesting ongoing research subject is also the application of the F-PRAM cost model to the message passing library programming. The cost model needs only minor changes, mostly on the secondary parameters. In practice, this would ease the estimation of the efficiency of MPI/PVM programs. Furthermore, by applying the F-PRAM parameters, the MPI/PVM programmer can write programs that adapt to different types of virtual parallel computers.

References

1. Batcher K.: Sorting Networks and their applications. *AFIPS Spring Joint Computing Conference* **32**, pages 307-314, 1968.
2. Cole R., Zajicek O.: The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 169-178, 1989.
3. Culler D. & al.: LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of 4th ACM Conference on Principles & Practices of Parallel Programming,* pages 1-12, 1993.
4. Gibbons P. B.: A More Practical PRAM Model. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 158-168, 1989.
5. Halstead R. H.: Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems* **7**,4, pages 501-538, October 1985.
6. Juvaste S.: *Reasoning of a Parallel Computation Model*. University of Joensuu, Department of Computer Science, Ph.Lic. Thesis, 1996. Available in Postscript from URL `http://cs.joensuu.fi/~sjuva/f-pram/lic.thesis.html`.
7. Juvaste S.: *Programming language FPM for F-PRAM*. University of Joensuu, Department of Computer Science, Report B-1997-X, to appear. Available in Postscript from URL `http://cs.joensuu.fi/~sjuva/f-pram/fpm.language.html`.
8. de la Torre P., Kruskal C.: Towards a Single Model of Efficient Computation in Real Parallel Machines. In *Proceedings of Parallel Languages, Europe*. LNCS 505, pages 6-24, 1991.
9. Valiant L. G.: A Bridging Model for Parallel Computation. *Communications of the ACM* **33**,8, pages 103-111, August 1990.
10. Veräjäntausta, J. 1997: *An F-PRAM emulator.* University of Joensuu, Department of Computer Science, Report B-1997-X, to appear.
11. Wirth N. B. 1982: *Programming in Modula-2*. Springer-Verlag. ISBN 0-387-12206-0.