A Realistic Cost Model for the Communication Time in Parallel Programms on Parallel Computers Using a Service Hardware*

Matthias Fischer[1] Jochen Rethmann[2] Alf Wachsmann[1]

[1]Universität-GH Paderborn
Heinz Nixdorf Institut und
Fachbereich Mathematik-Informatik
D-33095 Paderborn
email: {mafi,alf}@uni-paderborn.de

[2]Heinrich-Heine-Universität
Düsseldorf
Institut für Mathematik
Universitätsstr. 1
D-40225 Düsseldorf
email:
rethmann@cs.uni-duesseldorf.de

Abstract

In this report, we develop a cost model for the communication time on parallel computers consisting of processors and a service network, i.e., a network performing services like broadcast, synchronization, and global variables. Because we do not have a parallel computer at our disposal that is equipped with a service network, we emulate the service network on a reconfigurable Transputer network.

Our cost model describes the communication time of accesses to global variables and consists of a multi-linear function. The cost model includes the parameters packet size, send hot spot (the number of messages sent out by one processor), and number of processors accessing global variables. We show that these parameters influence the communication time in a high degree and capture important parameters like network contention.

We implement a Bitonic Sort, Sample Sort, Matrix Multiplication, and Connected Components algorithm, and we show that our model is able to predict the communication time within a 10% error if indirect service networks are used. The applications show that it is easy for a programmer to determine the parameter values for our model and that our new cost model precisely predicts the communication time of parallel algorithms.

We explore the interaction of hot spots and asynchrony and show that the influence of hot spots to the communication time is not as high as one would expect from theoretical considerations in a synchronous model. Therefore, we do not apprehend the hot spot in our cost model.

Furthermore, we minimize the communication time of accesses to global variables by finding a balance between the number of messages in the network and their size. Our model predicts the optimal values for these parameters which we validate by experiments. A modified implementation of our routing which determines on-line the optimal parameter values for an access to a global variable achieves good speed ups.

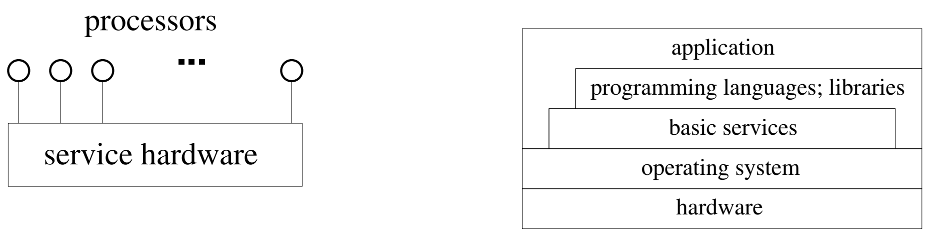
*Supported in part by the DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen."

Contents

1 Introduction

In this report, we introduce a realistic cost model for communication times of parallel programs. It is well known that the uniform cost model of PRAMs does not apply for todays' parallel computers. Therefore, many approaches try to overcome this problem. The new model considers blockwise communication, message latency, and other effects which can be observed on parallel computers.

Basic Services. The hardware we have in mind is a processor network which is connected via routing facilities. The routing strategies considered are mostly store-and-forward or wormhole routing. On top of this physical network, a software layer is built which realizes some useful *basic services* like routing, synchronization, and virtual shared memory in terms of global variables (see Figure 1). We use an extra network ("service hardware") for executing the service part of the hardware in contrast to [22], where the basic services and application programs are executed in the same network. "Service hardware" in this sense means a special purpose hardware to support standard processors with additional means in order to reduce their workload (see Figure 1).



(a) Hardware view. (b) Software view.

Figure 1: Concept of basic services.

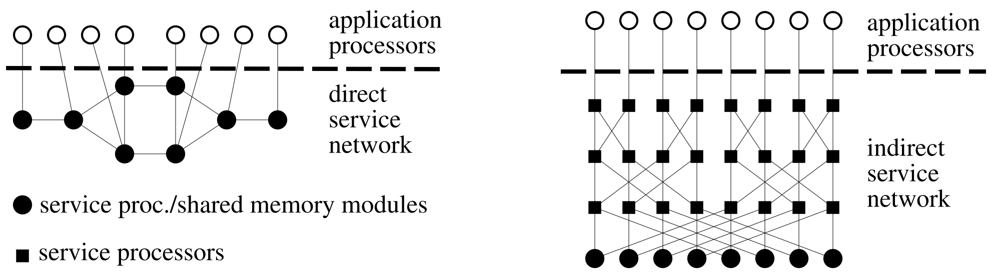
Examples for "service hardware" are the routing network of the GC/PP (Parsytec) where the standard processors (PowerPC 601) are connected by routing chips (T805) which are interconnected by a fat mesh. Other types of service hardware are the synchronization (and routing) network of the CM-5 (Thinking Machines) and the virtual shared memory hardware of the KSR1, the ALLCACHE engine (Kendall Square).

The idea of service functions also arises in software: there are several libraries and programming languages which support or realize service functions. Libraries that just support routing functions are MPI [19], the Parmacs-Macros [8], and PVM [12]. The Oxford BSP Library [20] is a library and OCCAM-light [23] is a high level programming language which both realize functions for routing, synchronization, and shared memory in terms of global variables.

Types of Service Networks. Network-based parallel computers can be divided into two classes depending on whether there is a processor and a memory module at each node of the network (direct network) or the processors and memory modules are interconnected by a network of switches (indirect network). We use the term *virtual shared memory* to refer to a memory that is distributed among distinct memory modules but, can be accessed by each processor via routing.

Our intention is to develop a cost model that allows precise predictions of the time needed to access shared memory in direct and indirect network-based machines. The parallel computer we had at our disposal is a Transputer system. Although the Transputer has routing devices with direct memory access (DMA) on-chip the CPU is involved every time a message has to be forwarded because buffering of messages and some other work have to be done by the CPU.

Because is it not easy to design service hardware for a parallel computer and to exchange it, we emulate this hardware on a processor network. Therefore, to get real autonomous routing devices, we split the processor network into two parts: one for executing the application program – we call this part *application network* – and one for simulating the service function – the *service network* (see Figure 2). We use direct service networks like the Cube-Connected-Cycles network (CCC) or the Shuffle-Exchange network (SE), and indirect service networks like the Butterfly network (BF). Throughout the paper we assume that the number of application processors is equal to the number of shared memory modules. In our indirect network, the service processors emulate the routing devices and the shared memory modules.



(a) direct service network: Shuffle-Exchange network of dimension three.

(b) indirect service network: Butterfly network of dimension three.

Figure 2: Different possibilities for simulating a service network on a processor network.

The algorithm we use to implement the basic service functions (routing, shared memory, and synchronization) on a store-and-forward Transputer system is a simple shortest paths routing which uses FIFO queues for storing the messages. In Section 2, we introduce the hardware we have in mind and show that the existing cost models can not be used in our case. We describe the hardware we use and the emulation of service hardware in Section 3.

Cost Model. There are two cost models, the BSP model by Valiant [21, 13, 14] and the LogP model by Culler et al. [9] which claim to be more realistic than the PRAM model. However, both models do not take some important parameters for the communication time into account. Making experiments on our system, it turns out that neither the BSP nor the LogP model can predict the runtime of programs realistically. Both models take an upper bound for the latency for sending messages of a fixed, short size. This assumption does not fit to our intended hardware.

We introduce k-k-accesses as basic communication pattern, where a portion r of the application processors are sending/receiving exactly k messages of length l in Section 4. For this routing pattern, we develop a cost function (with coefficients for each network topology) which depends on the three parameters k, l, and r. The used hard- and software implies a linear dependence of the function on each of these parameters which results in a trilinear function. We validate these considerations by performing measurements on our implemented system where we use the basic service functions to realize the k-k-accesses.

A *hot spot* at a shared memory module occurs if more than one processor access the same variable at the same time. These requests must be answered one after the other, so the computation of the processors is sequentialized. In Section 5, we show that the influence of hot spots on the runtime can be balanced out by asynchronous processor networks.

To demonstrate the precision of our model, we implement several application programs on top of our system (Section 6). We predict the communication times and compare them with the measured communication times. This comparison shows that our model is well suited for indirect service networks but lacks precision for direct service networks.

As an application of our model, we optimize the runtime of the routing we used for realizing the basic service functions in Section 7. For routing, there is a trade-off between the number of messages in the network and the lengths of these messages. We speed up the routing by splitting long packets into smaller ones such that the routing becomes more like wormhole routing. We use our cost model to calculate the optimal packet size in order to optimize the routing time. Measurements validate the predictions.

2 Motivation for a new cost model

A model should be well suited to handle the trade-off between applicability, precision, and generality, which means that one should be able to estimate the communication time of a program easily and precisely, whereas generality means that one should be able to describe many communication patterns in parallel programming. So, the task is to develop a model that finds a balance between detail and simplicity: all important parameters have to be taken into account in order to make precise predictions of communication times without making the model too complicated to handle.

2.1 Our hardware model

Our used hardware can be characterized as a processor network with a store-and-forward router. The processors are standard sequential processors. Each processor has at least one routing facility (router) which is interconnected in a network that can be described as an undirected graph.

The router works in a store-and-forward manner, i.e., a packet has to arrive completely on a router before it can be forwarded. Sending a packet to a direct neighbor takes a fixed amount of time (offset o) to build up the communication line. The transmission time for a packet is linear in the number of bits to be transmitted. Thus, the time to transmit a packet over a distance of d hops takes time

$$L_{s\&f} = d \cdot (o + s \cdot g) \ ,$$

with s beeing the packet size in bits and g the transmission time per bit. We assume that the constant offset o is small (so the transmission time depends linearly on the packet size) but can not be neglected. If we would change to wormhole routing, the influence of the distance between sending and receiving nodes becomes smaller. The transmission time for a worm, consisting of flits of length F bits, is

$$L_{wh} = o + s \cdot g + F \cdot g \cdot d \ .$$

A comparison of the store-and-forward routing and the wormhole routing shows that in both cases the transmission time for data linearly depends on one or more parameters. So, the store-and-forward router is no restriction for the generality of our cost model.

2.2 Existing models

PRAM model. The most popular but not realistic abstract parallel machine in theoretical computer science is the *PRAM* model. The advantages of the PRAM model are the shared memory and the synchronized working processors. Because of the uniform cost model, it is relatively easy to calculate the complexity of PRAM programs. Design of algorithms is relatively easy too, since all data structures are stored in the shared memory. On many real parallel machines, processors do not have shared memory but distributed memory and the processors work asynchronously. One consequence is that an access to distributed memory is much more expensive than access to local memory. Therefore, the uniform cost model of the PRAM is not able to predict the runtime of programs on real machines. The programming model of the PRAM, e.g., tending to use small global variables, is misleading for algorithm designer and programmers of real machines where rare use of communication can be (or is) essential.

PRAM extensions. There are several attempts to make the PRAM model more realistic. The *BPRAM* model [3] distinguishes between local memory and shared memory. Access to the shared memory is done by communication in blocks. This takes into account the fact that the costs for routing many small packets are larger than for routing few large packets. Another model which distinguishes between local and shared memory is the *LPRAM* model [2].

The *Phase-PRAM* model [15] tries to overcome the problem of asynchrony. The Phase-PRAM model divides the computation in phases. In each phase, processors work asynchronously. After each phase, all processors are synchronized.

The problem inherent to all PRAM extensions is to model the limited communication bandwidth and the send/receive overhead of communication. The send/receive overhead is the time a processor is engaged in the transmission or reception of a single message. During this time, the processor cannot perform other operations.

Distributed memory models. The problem of module contention, which occurs in processor networks, can be handled by the *Distributed Memory Machine* (DMM) [16] or the *Module Parallel Computer* (MPC) [18]. The memory is divided into memory modules. At any time step, the access to a single module is only allowed to one processor.

Today, the most popular realistic cost models are the BSP model and the LogP model but they do not scope the parallel computers described above in their communication time.

BSP model. Valiant's *Bulk Synchronous Parallel Model* [21] consists of processor/memory modules, a router, and a synchronization mechanism. The basic communication pattern considered is an h-relation, where certain processors send and receive up to h packets. The router is capable to route any h-relation in $g \cdot h + s$ steps, where g is the gap between two packets which have to be sent, s is the startup time, and h is the maximum number of packets which are sent or received by one of the processors. This model captures the limited communication bandwidth of real parallel machines by the parameter g. It further abstracts from the used network topology.

An extension of this model, called *BSP* model*, introduced in [6] includes an additional parameter for message lengths.

LogP model. Culler et al. suggested the *LogP model* [9]. It consists of processor/memory modules and an arbitrary kind of network to provide point-to-point communication. The network performance is described by three parameters L, o, and g, where L is the maximum latency, o the send/receive overhead, and g the gap between two packets which have to be sent. The aim of the LogP model is to find a balance between communication and local computation, such that idle times of the processors are minimized.

2.3 Drawbacks of the existing models

The LogP model is not suited to predict the execution time of algorithms precisely because, on some machines, the latency for routing packets strongly depends on the length of the path in the network. Therefore, estimating latency by the worst case (L) leads to non precise predictions. This is not crucial for those machines considered in [9] because the send/receive overhead in such machines is much greater than the latency L. This argument leads to the *LogGP model* [1] which uses the parameter G to model different packet sizes.

On machines like ours, the packet size does play an important role for the communication time, so the BSP and the LogP model are not suitable for our purposes. Also, the congestion (i.e., the communication load) of the network is a crucial parameter for the communication time. For the mentioned models (BSP, BSP*, LogP, LogGP), it is of no importance how many messages are contending in the network because they allways assume the worst case which produces in situations of low or medium load in the network great deviations of predicted and real communication time.

BSP and LogP model neglect important parameters because they do not model packet size and the number of communicating processors (i.e., the communication load) which are very important parameters that influence the latency in a large degree. We explain the two in the following.

If one processor has to access a fixed amount of data this can be done by sending one big or many small packets. On one side, too small packet size (i.e., many packets are sent) leads to a great overhead because each packet consists of the real data and a header and the time needed to transmit some data is not only determined by the transmission throughput but also by some small but not neglectable startup cost. On the other side, if the packet size is too large (i.e., few packets are sent) the packets can not be pipelined by the routing and the parallelism of the Transputer links is not utilized. Therefore, the aim is to find the optimum packet size dependent on the amount of data to access. In order to motivate this consideration we show an example and assume that one single processor sends one packet of size D to another processor over a distance of n hops.

The time needed to transmit D items over one link is $T_1 = S + D \cdot t$, where S denotes the startup cost and t the transmission time per item. If a packet has a distance of n links to travel the time for the transmission is $T_n = n \cdot T_1 = n \cdot (S + D \cdot t)$. If we split up the access into m accesses which produces m packets each of size $\frac{D}{m}$ the first packet needs time $n \cdot (S + \frac{D}{m} \cdot t)$ to reach the destination, while the last packet arrives at its destination after further $(m-1) \cdot (S + \frac{D}{m} \cdot t)$ steps, hence the total time of the split access is $T_{split} = (n + m - 1) \cdot (S + \frac{D}{m} \cdot t)$.

$$T_{split} < T_n \iff (n+m-1)(S + \frac{D}{m} \cdot t) < n \cdot (S + D \cdot t) \iff m < \frac{n-1}{S} \cdot D \cdot t$$

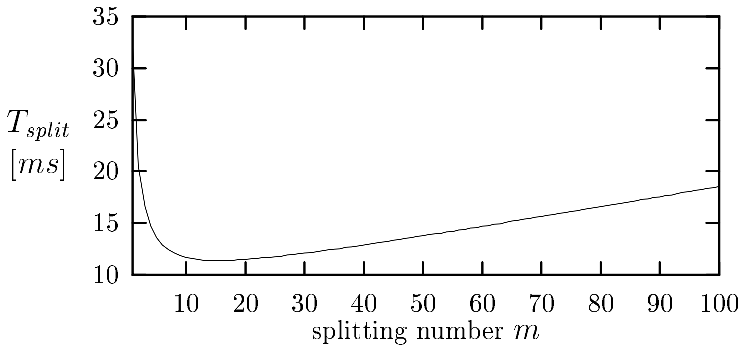


Figure 3: Influence of the splitting size m on the communication time. For $m = 1$ it is $T_{split} = T_n$.

The communication time T_{split} of the split access is less than the time T_n if $m < \frac{n-1}{S} \cdot D \cdot t$. Figure 3 shows the communication time if $S = 100\mu$s, $t = 0.8$ μs/item, $D = 10000$ items, and $n = 4$. The duration of the split access is greater than the duration of the unsplit access if $m > 240$, while the optimum number of packets is 15. Both, BSP and LogP model do not take these considerations into account. In Section 7 we show how one can determine the right size of the splitting number by the use of our cost model even if more than one processor accesses a global variable.

Another important influence on the routing time is the network contention. Communication costs in the case that exactly one pair of processors is communicating and in the case that many pairs of processors communicating are equal to $(2o + L)$ in the LogP model. On real machines the communication costs in the case that many pairs of processors communicate is more expensive than in the case that one pair communicate because in most routing situations contention occurs in the network. The LogP model neglect this fact because it always takes the worst case latency and hence has no parameter for the network contention.

3 Realization on a Transputer Network

Because we have no parallel computer at our disposal that is equipped with the appropriate routing facilities we have to emulate the service network. In this section, we describe the realization of the service network. First we describe the used hardware, second the process structure and give a more detailed description of the implementation. The tasks of the service processor (and processes) are routing and executing the service functions. We describe the processes for these tasks separately. Finally we compare pipelining in direct and indirect networks.

3.1 The Parsytec SC-320

The parallel computer we implement our emulation on is a Parsytec SC-320 [11]. It consists of a free configurable network of T800 Transputer processors (32bit RISC-processors). Each of the 320 processors has four routing devices on chip, so networks may have a degree of up to four.

A main difficulty in programming routing algorithms on Transputers is that the routing devices (links) transmit data in parallel by accessing the memory of the CPU directly (DMA). But they can not handle packets autonomously because the routing devices are not able to determine the destination of packets. So the CPU has to be involved each time a packet arrives. It has to manage buffering and to determine the destination of the packets. As a consequence a calculation running on a Transputer is interrupted if a packet wants to traverse the node. This is the reason why we only consider service networks (see Figure 2), in contrast to [22], where application programs and service functions are executed in the same network. Another speed reduction of the calculations is caused by the concurrent memory access of the processors and of the links, whenever the links transmit data.

Process communication is done via channels in a bit-serial fashion. In the case that processes are placed on different processors communication is done via Transputer links (hardware channels), while processes placed on the same processor communicate via software channels.

Summerizing, one can say that the Transputer network is a representative of the hardware model we described above.

3.2 Algorithms for the service functions and their implementation

There are 14 processes on a service processor (see Figure 4).

In order to avoid performance reduction each link must be supported by a process of its own (see Figure 4 “Distributor” and “Guarded output”). Only under this condition it is possible that links transmit data parallel.

Processes for routing. For routing packets in our service network we implemented a shortest path routing algorithm. This routing algorithm is not free of deadlocks. If necessary, packets that arrive at a processor are stored in queues before they are forwarded to a neighboring processor. The sizes of these queues are large enough to avoid deadlocks in almost every case in practice. In a preprocessing phase, before starting the application program, the shortest paths of the service network are computed sequentially and stored in a look-up table.

The theoretical behavior of the shortest path algorithm is unknown. In [4] a routing is presented which uses shortest paths as routing paths, but unbounded queues and a growing rank protocol is used. It is shown that routing any set of packets along shortest paths

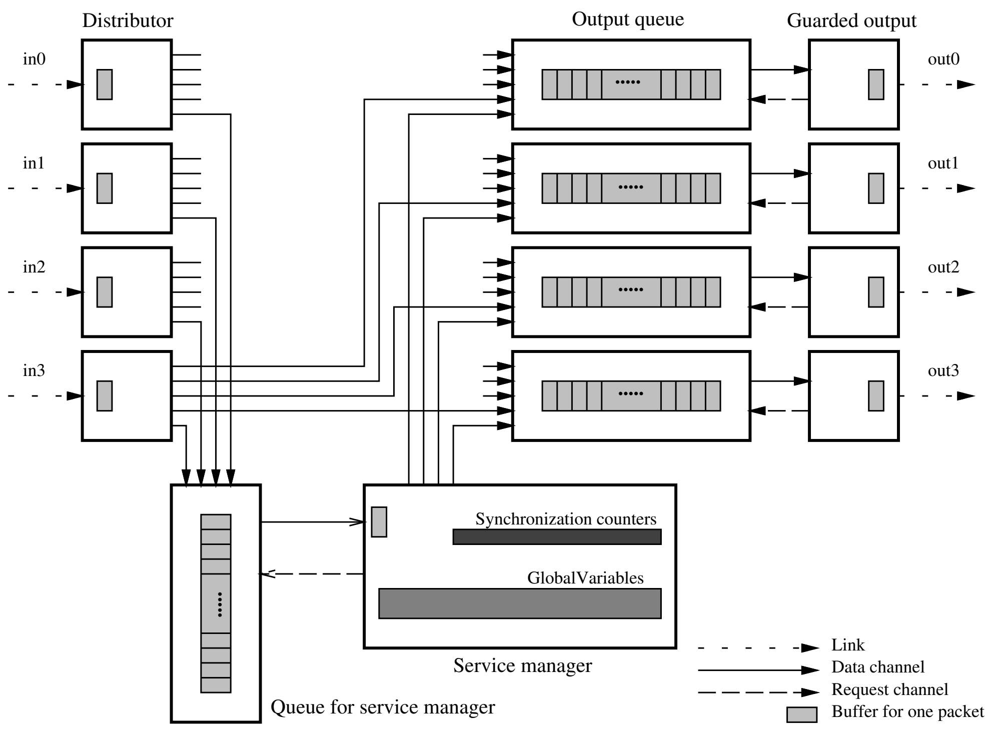


Figure 4: Process diagram of the service processor.

through an arbitrary n-processors network takes time $O(\text{congestion} + \log n + \text{diameter})$ with high probability.

The process “Distributor” (see Figure 4) is connected with an incoming link of the Transputer and it processes incoming packets. If an incoming packet reaches its destination processor the Distributor sends the packet to the queue of the service manager. If the packet has not reached the destination processor the Distributor searches in the look-up table for the correct outgoing link and sends the packet to the according queue and guarded output.

The process “Queue” buffers packets in a FIFO queue. A further task is to combine packets for the same synchronization operation. For this the FIFO queue searches for a packet of the same synchronization operation and combines them if other packets are found.

The Transputer T800 does not support guarded outputs. In order to avoid a blocking of the queues, because of packets which cannot be delivered to a neighbor processor, the process “Guarded output” simulates this type of channel. Each process “Guarded output” is connected with one outgoing link and can buffer one packet.

Processes for executing the service functions. We realize shared memory in terms of global variables which may have an arbitrary type (e.g., integer, arrays, etc.). The variables get an unique number and are distributed over the service network either via universal hashing or via direct placement.

We have to distinguish read and write accesses to global variables. If an application processor wants to read a global variable a read request is sent to the service processor that stores the global variable. The service processor writes the contents of the variable into a packet and sends it back to the requesting application processor. If an application processor wants to write some data into a global variable it writes the data into a packet and sends it to the service processor that holds the global variable. The service processor writes the data into the global variable and sends an acknowledgement back to the application processor. For the synchronization of all or parts of the processors we implemented a barrier synchronization mechanism.

The service functions are executed by the process “Service manager”. This process manages the counter for synchronization and the memory for global variables. According to the header of the packet (at most 12 Bytes) it detects the type of service function. The service manager watches the counter for synchronization and starts a broadcast for an acknowledgement, if necessary. For accesses to global variables it copies the data from the global memory into a packet or from a packet into the global memory.

Differences to routing processors of indirect service networks. Indirect service networks are consisting of routing processors and service processors (see Figure 1). The only task of the routing processor is to route packets. It does not store synchronization registers or global variables. So the process structure is similar to the process diagram of the service processor without the process “Service manager” and the queue for the service manager.

4 The new cost model

In order to predict the runtime of the basic service function we have to define a communication pattern that we want to perform and whose runtime we want to model. The pattern should be as usable but should have more strength than the h-relation of the BSP model which does not distinguish send and receive hot spots: A h-relation is given if one processor sends up to h messages to distinct processors and if up to h processors sending one message to the same processor. As we see later in this section the first type of hot spots influences the communication time in a large degree in contrast to the latter type.

Definition 1 (k-access) *Let $[p]$ denote the set $\{0,\ldots,p-1\}$ and $F_k = \{f \mid f : [p]\times[k] \longrightarrow [p]\}$ as $k \in \mathbb{N}$. Every processor P_i contains k packets, labeled by $0,\ldots,k-1$. “Routing the function $f \in F_k$” means: Send the j-th packet from processor P_i to the shared memory module $S_{f(i,j)}$, access the global variable, and send back a packet to processor P_i. All k packets of a processor are sent successively in k* steps *one after another.*

In the model presented here we look at the more special k-k-accesses of the next definition.

Definition 2 (k-k-access) *Let $f \in F_k$. Let us abbreviate $f(i,j_0)$ by $f_{j_0}(i)$ for any fixed $j_0 \in [k]$. If $f_{j_0} : [p] \longrightarrow [p]$ is a permutation for all $j_0 \in [k]$, then this special case of a F_k routing function is called k-k-access.*

Note: If only a portion $r < 1$ of the processors are involved in a k-k-access than f_{j_0} is a partial permutation for all $j_0 \in [k]$. We have to distinguish between hot spots at the shared memory modules and at the application processors. We call the first *receive hot spot* and the latter *send hot spot*. For example the case where only one processor accesses n global variables at the same time (i.e., a send hot spot) is described by $r = \frac{1}{p}$ and $k = n$.

In the following we describe the dependencies of the communication time for a k-k-access on different parameters, how we model these dependencies, and how we validate our considerations by measurements. To determine the coefficients of our cost function we measure the runtimes for several different k-k-accesses and average them. For each k-k-access we choose k permutations at random. In our model the contention is captured on the one side by the parameter r, on the other side by building the averages on many different access patterns.

Packet size *l*. Our service network is a processor network with a store-and-forward router. Therefore, transmission of packets from one processor to a neighbor processor via a link depends linearly on the packet size. Packets are buffered in FIFO queues and routing is done with the help of look-up tables. So processors internal transport of packets depends linearly on the packet size, too. Thus, we expect a linear dependency of the communication time of a k-k-access on the packet size l which is confirmed by our measurements (cf. Figure 5).

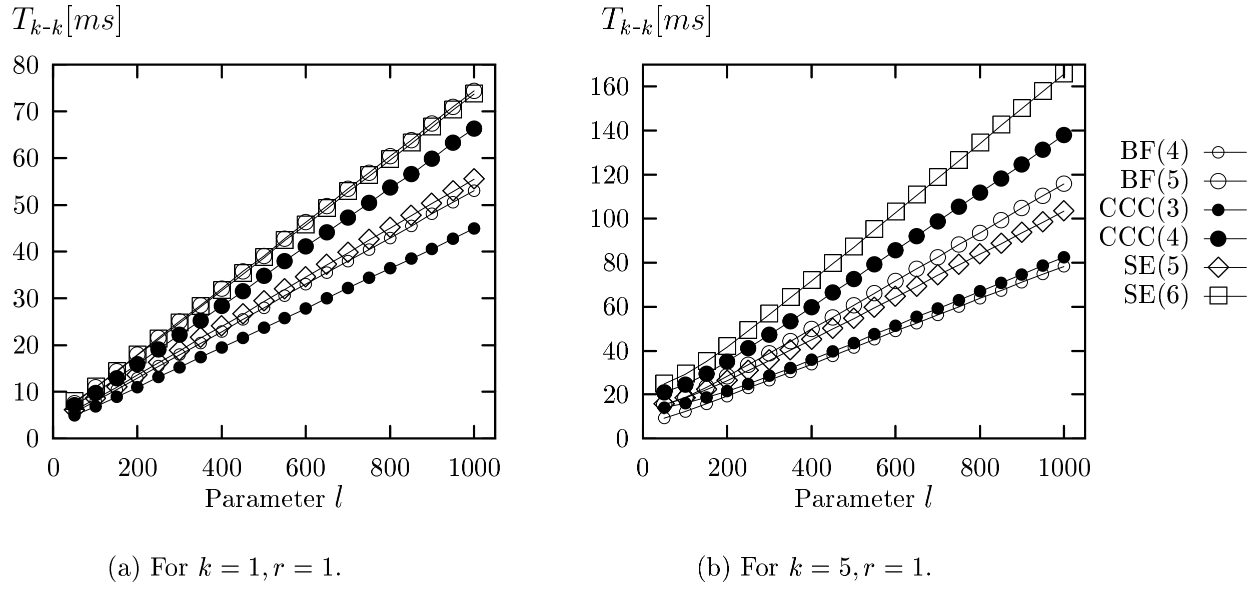


Figure 5: Dependency on parameter packet size l of the communication time of a k-k-access.

Each processor on the way of a packet through the service network causes an offset which is produced through initialization of variables, initialization of links for transmission, and the packet header. This results in an offset which depends on the average path lengths of the network (cf. Figure 5).

The beginnings of the curves (cf. Figure 5) are bended. Small packets hinder others not as much as large packets do because large packets stay longer in a network node than small packets. This leads to larger routing times per Bits for larger packets.

Portion *r* of accessing processors. It is very difficult to determine the network contention for an asynchronous processor network. In order to respect the contention by a parameter we introduce the parameter r which is that portion of processors that is involved in a k-k-access to global variables. We call an application processor that accesses a global variable an “accessing processor”. The processors that do not call service functions compute local operations or just wait.

The parameter r represents the behavior of the network in the case of contention. If the number of accessing processors is small then the contention in the service network is small. More accessing processors are causing higher contention because the network contains more packets and the probability that two packets arrive at the same processor at the same time is high. This leads to longer execution times for a k-k-access.

Although the greedy routing algorithm of an one-to-one routing problem in the Butterfly network takes at least $O(\sqrt{N})$ routing steps in worst case the routing algorithm performs quite well on average [17]. Thus, the contention in the Butterfly network is small on average. Since our measured access times are average values we expect that the dependency of the runtime of a k-k-access on r is very small for indirect service networks like Butterfly networks (BF).

In Figure 6 we see that for direct service networks like Cube-Connected-Cycles (CCC) or Shuffle-Exchange (SE) networks the access time increases much more, so the contention is higher in direct networks than in indirect networks.

Furthermore, an indirect network contains more processors than a corresponding direct network: In the case of a direct service network the number of packets for a k-k-access is $r \cdot k$ packets per processor on average. In the case of an indirect service networks with $p \cdot \log p$ processors there are $r \cdot \frac{k}{\log p}$ packets per processor on average.

Because of our measurements (cf. Figure 6) we model execution time of a k-k-access in dependency of r by a linear function.

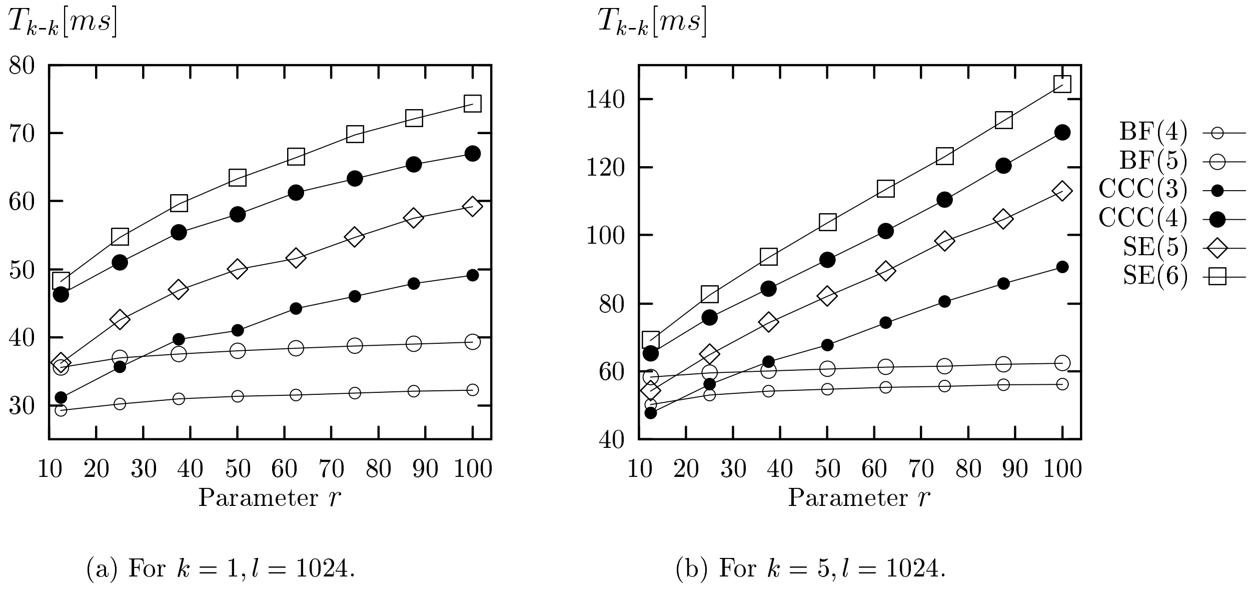
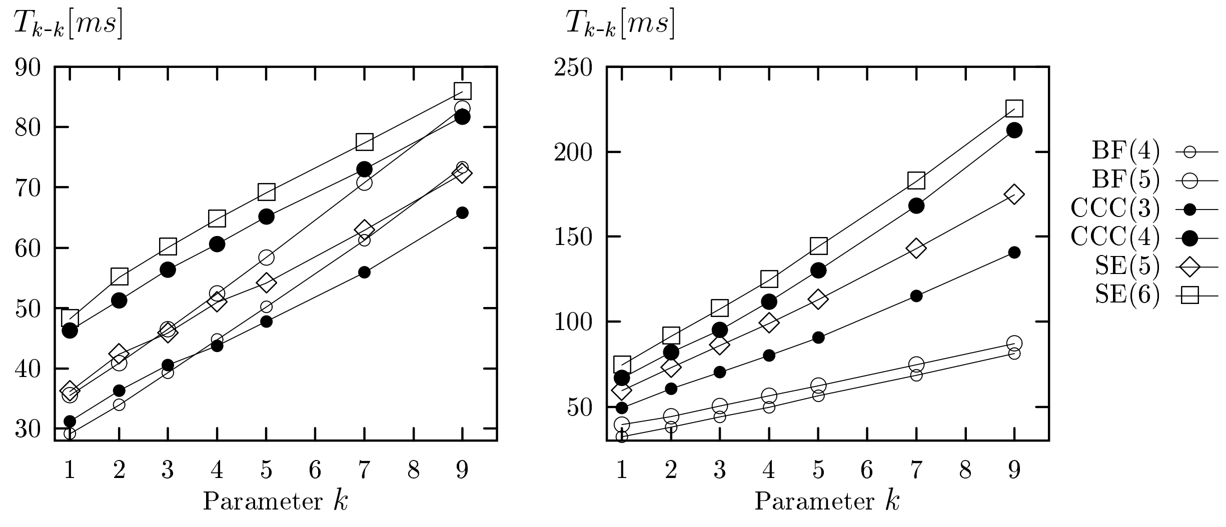


Figure 6: Dependency on parameter portion of accessing processors r of the communication time of a k-k-access.

Send hot spot k. Parameter k is the number of packets which are sent/received in one k-k-access by each accessing application processor. If all application processors are involved in a k-k-access ($r = 1$) then each shared memory module has to process exactly k packets. If r is less than 1 the shared memory modules have to process at most k packets.

We expect a linear dependency of the runtime of a k-k-access on k because each of the involved application processors sends/receives exactly k packets one after another. This results in a send hot spot of size k at the application processors. These considerations are confirmed by our measurements (cf. Figure 7).



(a) For $r = \frac{1}{8}, l = 1024$.

(b) For $r = 1, l = 1024$.

Figure 7: Dependency on parameter size of send hot spots k of the communication time of a k-k-access.

Cost function. We denote the parallel execution time of a k-k-access by the function $T_{k\text{-}k}(k, l, r)$. Several measurements with different values for the parameters show that the execution time of a k-k-access can be modeled precisely by a trilinear function. The coefficients $c_0, \ldots, c_7$ depend on the type of network used and the size of the network.

$$T_{k\text{-}k}(k, l, r) = c_7 \cdot klr + c_6 \cdot kr + c_5 \cdot kl + c_4 \cdot lr + c_3 \cdot k + c_2 \cdot l + c_1 \cdot r + c_0$$

The coefficients are quantified with the method of least square. The measured coefficients for some networks are listed in Table 1.

network	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0
BF(3)	0.519	102.2	4.989	2.341	382.4	15.23	-51.73	1242
BF(4)	0.475	108.8	5.203	2.760	371.9	20.78	-27.32	1619
BF(5)	0.056	111.0	5.419	3.583	373.5	26.49	-1.924	1969
SE(3)	3.371	715.5	2.304	11.22	312.4	9.413	-51.72	1242
SE(4)	6.896	1198	2.569	17.13	199.0	21.55	-710.0	1765
SE(5)	10.28	1852	2.205	10.52	90.98	32.24	-1388	2426
SE(6)	14.62	2321	1.879	7.360	86.71	44.53	-1776	2986
CCC(3)	7.631	1051	2.401	9.169	239.0	26.75	-579.7	1927
CCC(4)	14.53	1983	1.398	-0.285	159.9	43.23	-1335	2681

Table 1: Coefficients for the cost function of the communication time of k-k-access for some networks

Pipelining in direct and indirect networks. By varying the parameters k and r we confirm that pipelining in leveled networks (like the Butterfly networks) is better than in direct networks. In a direct network with average path length d sending k packets from one processor to another processor costs time $\Omega(k + d)$ in average case because the first packet reaches its destination after d hops while the last packet arrives at the destination after further $k - 1$ steps (pipelining). If each processor sends k packets to another (like in a permutation pattern) it costs $\Omega(kd)$. In each step only p packets can be sent because each of the p processors can transmit at most one packet at a time, but $p \cdot k \cdot d$ steps have to be executed. In this case the pipelining effect is very small.

If the processors at the first level of an indirect network (like BF) send k packets to the processors at the last level there is pipelining, too. Then $p \cdot k$ packets have to be sent over a distance of $\log(p)$ hops, so $p \cdot k \cdot \log(p)$ steps must be executed. In each step (except the first $\log(p)$ and the last $\log(p)$ steps) $p \cdot \log(p)$ packets can be sent by the routing switches. Communication time in comparison to the case that only one processor sends k packets is nearly the same on the average.

Measurements confirm this theoretical consideration (cf. Figure 8). Figure 8 shows an example for an indirect network (Butterfly) and a direct network (Shuffle-Exchange). We compare two cases: In the case $r = 1$ all processors are involved in a k-k-access and in the case $r = 1/8$ only two processors of the 16 processors network are involved in a k-k-access. The differences of $T_{k\text{-}k}$ of these two cases for the Butterfly network are very small, for the Shuffle-Exchange they are very high.

5 Receive hot spots and asynchrony

Until now, we regarded only k-k-accesses. In order to quantify receive hot spots we now introduce a new parameter h and use the more general k-accesses which are defined in Section 4. The value of parameter h is a real number in the range $[\frac{1}{p} \ldots 1]$ and describes the maximal portion of application processors which access in at least one of the k steps of a k-access the same shared memory module. In the case of k-k-accesses we have $h = \frac{1}{p}$. If h is equal to 1 then in at least one of the k steps of a k-access all application processors access the same shared memory module. In Section 5.1 we show that the influence of receive hot spots on the communication time is not as large as one might expect by theoretical considerations.

We show that the communication time is influenced in a high degree by the effect of asynchrony. By asynchrony we mean the "range" between two successive synchronizations. We quantify this range by introducing the new parameter a. The value of parameter a is the number of k-accesses which follow one after the other without synchronization or local computation. In other words: The value of a is the number of k-accesses between to successive synchronizations.

Let the ratio T_k/a denote the *amortized communication time* for one of the a successive k-accesses. In Section 5.2, we show that the influence of receive hot spots (i.e., $h > \frac{1}{p}$) on the amortized communication time decreases if the value of parameter a increases. This means, if many hot spot k-accesses are executed one after another without synchronization between the accesses, the amortized communication time for one of these accesses is smaller than for the execution of a few hot spot accesses one after another.

In Section 5.3 we show that the previously described effect can be observed even in the case of k-k-accesses (i.e., $h = \frac{1}{p}$). We show that in general the amortized communication time decreases if many k-accesses are executed one after another (i.e., the value of parameter a is large). But we will see that a saturation occurs for larger values of parameter a that means a further increase of the value does not lead to a further reduction of the amortized communication time.

5.1 The influence of hot spots on the communication time

A hot spot at a shared memory module occurs if more than one application processor access the same variable (or different variables at the same shared memory module) at the same time. These requests have to be answered one after the other and so the runtime for such a memory access depends linearly on the number of accessing processors. This is the case in synchronous indirect service networks where all processors are equally distant from the shared memory modules and the starting times of the packets are equal.

In asynchronous processor networks and direct networks the effect of hot spots is different and can be neglected under certain conditions. Figure 9 shows the measurements of the communication time over the parameter h for two networks and two values of parameter k. We begin our study of hot spots in the case $a = 1$, i.e., between two successive k-accesses there is always a synchronization. If all p application processors (e.g., $p = 32$ in the BF(5)) accesses the same global variable ($h = 1$) than the communication time is not 32 times as large as it is in the case of k-k-accesses (i.e., $h = \frac{1}{p}$) the slow down is at most two.

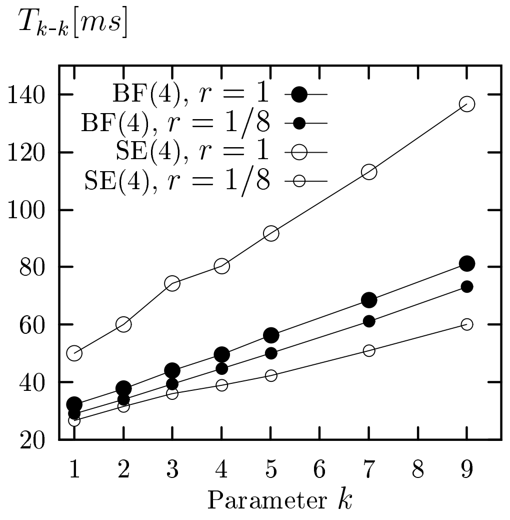


Figure 8: Dependency on parameter size of send hot spots k of the communication time of a k-access, $l = 1024$.

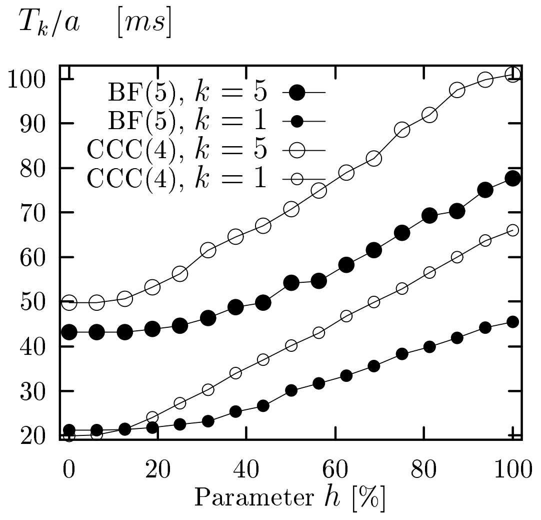


Figure 9: Dependency on parameter h of the communication time of a k-access, $l = 256$, $a = 1$.

Let us first take a look at indirect service networks (like BF). Assume that each accessing processor sends only one packet (i.e., $k = 1$) and that the starting times are different because of the asynchrony. If only few application processors send packets to the same shared memory module (i.e., h is small) the arrival time for the packets differ because of the different starting times and the equal distance of the processors from the shared memory modules, so no real hot spot arise. If many application processors send packets to the same shared memory modules (i.e., h is large) some packets arrive nearly at the same time because the starting times of the packets usually do not differ in a wide range. So the linear dependency of the runtime on the number of packets starts at a certain amount of accessing processors (cf. Figure 9).

Now let us take a look at direct service networks (like CCC and SE). Assume that the starting times are all equal and each accessing processor sends only one message (i.e., $k = 1$). If only few application processors access global variables the packets arrive at different times at the shared memory module because the processors are not equally distant from the shared memory module. No real hot spot arise. If many application processors access the same global variable then some packets arrive at the same time because not all the paths have different length. As in the case of indirect networks the linear dependency starts at a certain amount of accessing processors in direct service networks, too.

The influence of the different path length can be balanced out by different starting times. If a packet that has a long distance to travel is sent before a packet that has a short distance to go the packets can arrive at the same time, so the influence of receive hot spots on the communication time in direct service networks is stronger than in indirect service networks (cf. Figure 9).

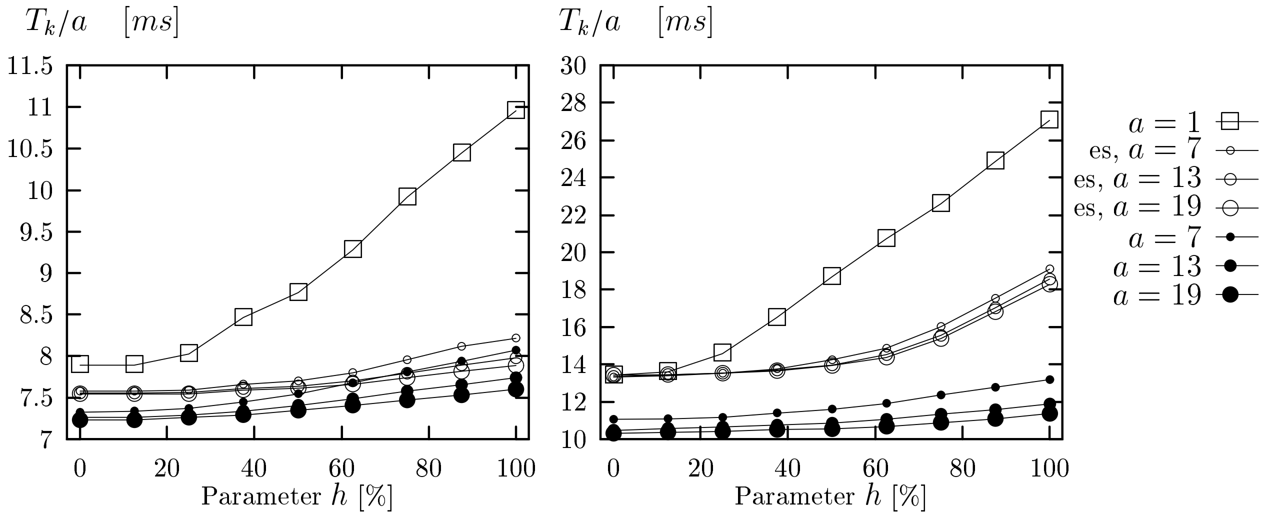
If k is greater than 1 the described effect occurs simultaneously at several shared memory modules, thus the explanation holds in this case, too.

Two scenarios. Now we extend our study to the case $a > 1$, i.e., there is no synchronization between successive k-accesses. In the first scenario we involve in each of the a k-accesses the same shared memory module (i.e.., the hot spots are the same in each of the a k-accesses). In the diagrams we denote this case with the prefix "es". In the second scenario we change in each of the a k-accesses the accessed shared memory modules (i.e.. the hot spot changes from one of the a accesses to the next one). We denote it with the lack of "es". Our intention for the first scenario is to produce a stronger type of hot spot situation than in the second scenario.

5.2 Influence of asynchrony on the hot spot effect

We show that large values of parameter a reduce the influence of hot spots on the amortized communication time. Let us first take a look at indirect service networks (like BF) and the simple case $k = 1$. Assume that the first 1-access is done at the same time by all application processors and each application processor do a successive 1-accesses without intermediate synchronizations. Then the starting times of all packets except the first ones of each application processor are shifted. All the destinations are equidistant from the sources of the packets, so the first packets of each application processor of a k-k-access arrive nearly at the same time at the shared memory module. The answers for these packets are sequentialized at the module, so that the last acknowledgement or the last data packet arrive at slightly shifted times at the requesting application processor and the next packet of this application processor is injected into the network with a shifted starting time. This effect leads to asynchrony and prevents a hot spot at the shared memory module for the following packets. So only the first packet of each application processor produces a receive hot spot.

In direct service networks (like CCC and SE) the shifted starting times of the packets do not necessarily cause different arrival times for the following packets because the different path lengths can balance out the shifted starting times. Figure 10 shows measured curves of the amortized communication time in dependency on several values of parameter h. The amortized communication time decreases if the value of parameter a increases.



(a) Butterfly network of dimension 3. (b) Cube-Connected-Cycles network of dimension 3.

Figure 10: Dependency on parameter h of the communication time for a k-access, $l = 256, k = 1$.

If k is greater than 1 the described effect occurs simultaneously at several shared memory modules, thus the explanation holds in this case, too.

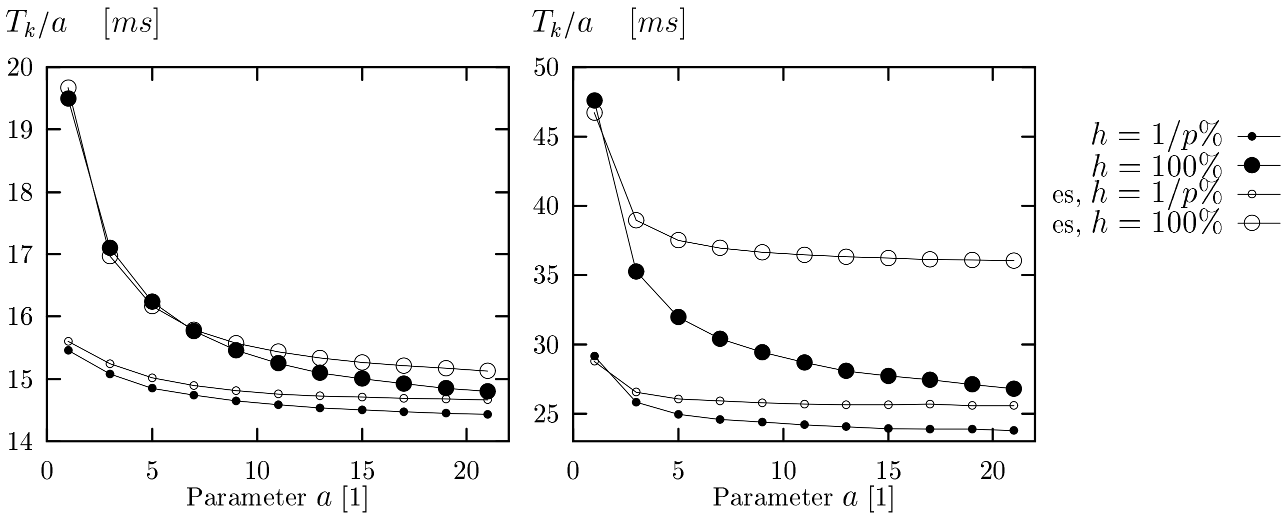
If we change the destinations of the hot spots in each of the a k-accesses the hot spot effect decreases (cf. curves of Figure 10 without "es"). Because of the time shifting packets of former k-accesses cannot meet packets of the next k-access at the same shared memory module.

5.3 Reduction of the communication time by large asynchrony

In this section we show that a large value of parameter a reduces the amortized communication time in the case $h > \frac{1}{p}$ (i.e., k-accesses) as well as in the case $h = \frac{1}{p}$ (i.e., k-k-accesses). Figure 11 shows measurements for the Cube-Connected-Cycles and the Butterfly network. The figures show the measured amortized communication time in dependency of different values of parameter a. For each network we show the two cases of no hot spots ($h = \frac{1}{p}$) and a hot spot in which all application processors are accessing one service processor ($h = 1$).

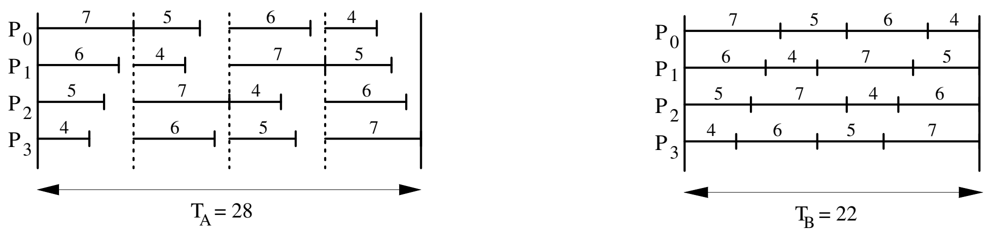
The amortized communication times decreases for all the curves if the value of the parameter a increases but all curves reach a saturation point (e.g., $a = 7$ for the Cube-Connected-Cycles network in the case of no hot spots, $h = \frac{1}{p}$).

To explain the decreasing communication time let us regard two situations (see Figure 12). In the first situation each application processor executes z successive accesses with a synchronization after each access (Figure 12(a), $z = 4$). In terms of our model we have z k-accesses with the parameter $a = 1$. In the second situation each application processor executes z successive k-accesses without intermediate synchronizations (Figure 12(b), $z = 4$). In terms of our model we have z k-accesses with parameter $a = z$. We denote the communication time



(a) Butterfly network of dimension 3. (b) Cube-Connected-Cycles network of dimension 3.

Figure 11: Dependency on parameter a of the communication time for a k-access, $l = 256, k = 5$.



(a) Case A: Four k-accesses with an synchronization after each k-access ($a = 1$).

(b) Case B: Four successive k-accesses without intermediate synchronization ($a = 4$).

Figure 12: Four processors executing four k-accesses. The numbers are the communication times for one k-access.

for the first situation with T_A and for the second with T_B. Because mostly the maximum communication time for each of the z k-accesses is reached by another application processor we can state that $T_A > T_B$. This effect is illustrated in Figure 12. The maximum time for each of the z k-accesses is reached first by application processor P_0 then by P_2, P_1, and P_3. Because of the missing synchronizations in the second situation, application processors which have finished their k-access can start immediately the following k-access. So idle time is reduced and the communication time shrinks. This is the reason why the amortized communication times is large for small values of parameter a in Figure 11.

6 Precision of the model

To determine the precision of our new model we compare the predicted with the measured communication times for several implemented algorithms. It turns out that one can handle the model very easily. Furthermore, the model is a good framework for algorithm design and it is not restricted to a few algorithmic techniques. In the following we describe the algorithms, situations in which the prediction error is very small, and situations in which the error is large. We analyze the reasons for these errors and draw conclusions how to improve the model.

The errors we talk about are mean values over different packet sizes, input sizes, and send hot spots. Our measurements show that indirect service networks are described very precisely by the model. In all but one example we obtain errors less than 10%. The inaccuracy between predicted and measured communication times in direct service networks is much greater and can be explained by three major influences: synchrony, data locality, and communication patterns.

6.1 Matrix Multiplication

Given two $n \times n$ matrices $A = (a_{ij}),\ B = (b_{ij})$, compute the product $C = (c_{ij}) = A \cdot B$ of the two matrices with p application processors. For simplicity we assume that p divides n. We use one-dimensional arrays of size $n \cdot \frac{n}{p}$ to store $\frac{n}{p}$ rows of matrix A and C and $\frac{n}{p}$ columns of matrix B in each of the p shared memory modules and we distribute the submatrices evenly over the shared memory modules.

More precisely, we divide the matrices into p submatrices each: $A = (A_0 A_1 \ldots A_{p-1})^T$, $B = (B_0 B_1 \ldots B_{p-1})$ and $C = (C_0 C_1 \ldots C_{p-1})^T$. To compute the submatrix C_i one has to multiply submatrix A_i with all the submatrices of B. By multiplying submatrix A_i and B_j we get the $\frac{n}{p} \times \frac{n}{p}$ submatrix $C_{ij} = A_i \cdot B_j$ of C_i while $C_i = (C_{i0} C_{i1} \ldots C_{ip-1})$. To avoid receive hot spots each application processor starts its multiplication with a different submatrix of B or in other words, each application processor accesses the submatrices of B in a different order.

```
Program of application processor $i$:
read submatrix $A_i$
for $j := i+1$ to $i+p$ do
    synchronize all application processor   (*)
    read submatrix $B_{j \bmod p}$
    compute $C_{ij \bmod p} = A_i \cdot B_{j \bmod p}$ of submatrix $C_i$
write submatrix $C_i$
```

Line (*) of the algorithm is not necessary and can therefore be left out. But this algorithm is well suited to show the influence of synchrony on the precision of the model, so please keep this point in mind, we will refer to this later on.

Let l be the packet size and k the size of the send hot spots. To compute the product of two $n \times n$ matrices the algorithm has to read or write $p+2$ times a submatrix of size $n \cdot n/p$. The amount of n^2/p matrix elements of each submatrix is accessed by packets of size l while each k-k-access delivers $k \cdot l$ matrix elements. So it needs $n^2/(p \cdot k \cdot l)$ k-k-accesses to read or write one submatrix. Thus, the parameters k and l can be fixed by the user (or by the system) and they are not dependent on the algorithm. The communication time can be estimated by the term

$$T_{\mathrm{MM}}(k,l) = (p+2) \cdot \left\lceil \frac{n^2/p}{l \cdot k} \right\rceil \cdot T_{k\text{-}k}(k,l,r=1) \ .$$

6.2 Computing Connected Components in Dense Graphs

Given an undirected graph $G = (V, E)$ with n vertices and $m = \Theta(n^2)$ edges, compute the connected components of G with p processors. For simplicity we assume that p divides n and p is a power of 2. We use two-dimensional arrays of size $n \times \frac{n}{p}$ to store $\frac{n}{p}$ rows of the adjacency matrix A of G in each of the p shared memory modules, hence the blocks of rows are equally distributed over the shared memory modules. The algorithm we use to compute the connected components is due to Woo and Sahni [24]. In the first step each application processor i computes a spanning forest W_i based on the information of the partition of the adjacency matrix stored in shared memory module i, which we denote with A_i. This is done by a breadth-first search. The spanning forests define a relationship R between pairs of vertices $v_i, v_j \in V$:
$v_i \; R \; v_j \iff v_i, v_j$ are in the same tree in at least one forest. The equivalence classes of this relationship represents the connected components of the graph G. In the second step of the algorithm these equivalence classes are computed by merging the spanning forests.

```
Program of application processor $i$:
read submatrix $A_i$ of the adjacency matrix $A$
Breadth-First Search on subgraph $G_i$ described by submatrix $A_i$
write the computed forest $W_i$
synchronize all application processors
for $j := 1$ to $\log(p)$ do
    if $i \bmod 2^j = 0$ then
        read the forest $W_{i+2^{j-1}}$
        compute the new forests $W_i$ by merging the old forest $W_i$ and $W_{i+2^{j-1}}$
        write the new forest $W_i$
        synchronize all accessing application processors
```

At the end of the computation the connected components are stored in the forest W_0. Let l be the packet size and k the send hot spot. The access to one part of the adjacency matrix of size $\frac{n^2}{p \cdot 16}$ in the first step of the algorithm (16 edges are stored in one variable of type "integer") can be done in time

$$T_{\text{bfs}}(k,l) = \left\lceil \frac{n^2/(p \cdot 16)}{k \cdot l} \right\rceil \cdot T_{k\text{-}k}(k,l,r=1) \; .$$

Application Processor 0 has to access $\log(p)$ times the spanning forests which are computed by other application processors. At each step there are only half the application processors accessing global variables than in the previous step. All the accesses to spanning forests (of size n) can be done in time

$$T_{\text{merge}}(k,l) = \left\lceil \frac{n}{k \cdot l} \right\rceil T_{k\text{-}k}(k,l,r=1) + 2 \cdot \sum_{i=1}^{\log(p)} \left\lceil \frac{n}{k \cdot l} \right\rceil \cdot T_{k\text{-}k}(k,l,r=1/2^i) \; .$$

The total communication time of this algorithm is $T_{\text{bfs}}(k,l) + T_{\text{merge}}(k,l)$.

6.3 Bitonic Sort

The algorithm we use to sort N numbers (or keys) is based on Batcher's sorting network [5] and is due to Culler et al. [9]. It is based on repeatedly merging two bitonic sequences to form a larger bitonic sequence.

Let the p application processors and the p shared memory modules be numbered with $0, \dots, p-1$.The basic operation of sorting networks is to compare two keys and exchange these if a given condition holds (*Compare/Exchange-operation*). To adapt this technique to our service network let us assume that shared memory module A holds the keys $x_1, \dots, x_n$ and shared memory module B holds the keys $y_1, \dots, y_n$. Then application processor A reads all the keys in shared memory module B and vice versa. From a previous step current values of the keys of shared memory module A (B) are stored locally in application processor A (B). Both application processors compare the pairs x_i and y_i, $i = 1, \dots, n$, and store the maximum or the minimum of these pairs in the according shared memory module. Let us use the following abbreviations: $\text{swapmax}(X, Y) := (\max(x_1, y_1), \dots, \max(x_n, y_n))$ and $\text{swapmin}(X, Y) := (\min(x_1, y_1), \dots, \min(x_n, y_n))$.

Let $\text{bit}(i, b)$ be the value of bit b_i in the binary representation of $b = b_{\lceil\log(b)\rceil}, \dots, b_i, \dots, b_0$. The local sorting of n keys on each application processor is done by "radix sort". We use one-dimensional arrays A_i of size n to store the keys and distribute them evenly over the shared memory modules. Let $\oplus$ denote the exclusive-or function.

```
Program of application processor $i$:
read the keys of the global array $A_i$
if bit(0,$i$) = 0 then sort local $(x_1, \dots, x_n)$ in increasing order
                  else sort local $(x_1, \dots, x_n)$ in decreasing order
for $j := 1$ to log($p$) do
    for $s := (j-1)$ downto 0 do
        synchronize all application processors
        write the keys $(x_1, \dots, x_n)$ into the global array $A_i$
        synchronize all application processors
        read the keys $(y_1, \dots, y_n)$ from the global array $A_s$
        if ( $\text{bit}(s,i) \oplus \text{bit}(j,i)$ ) = 1
           then $(x_1, \dots, x_n) := \text{swapmax}((x_1, \dots, x_n), (y_1, \dots, y_n))$
           else $(x_1, \dots, x_n) := \text{swapmin}((x_1, \dots, x_n), (y_1, \dots, y_n))$
    if bit($j$,$i$) = 0 then sort local $(x_1, \dots, x_n)$ in increasing order
                      else sort local $(x_1, \dots, x_n)$ in decreasing order
synchronize all application processors
write the keys $(x_1, \dots, x_n)$ into the global array $A_i$
```

Let l be the packet size and k the send hot spot. All application processors are involved in each k-k-access, therefore, r is equal to one. The communication time can now be estimated by the term

$$\begin{aligned} T_{\text{bitonic}}(k, l) &= 2 \cdot \left\lceil \frac{n}{k \cdot l} \right\rceil \cdot T_{k\text{-}k}(k, l, r) + \sum_{i=1}^{\log(p)} 2i \cdot \left\lceil \frac{n}{k \cdot l} \right\rceil \cdot T_{k\text{-}k}(k, l, r) \\ &= (2 + \log^2(p) + \log(p)) \cdot \left\lceil \frac{n}{k \cdot l} \right\rceil \cdot T_{k\text{-}k}(k, l, r) \ . \end{aligned}$$

6.4 Sample Sort

Sample sort or splitter sort is a randomized sorting algorithm due to Blelloch, Leiserson, and Maggs [7]. For simplicity we assume that $N = n \cdot p$ for some integer n. The basic idea of this sorting algorithm is to split the N numbers (or keys) into p sequences $S_0, \ldots, S_{p-1}$, such that the sequences have approximately the same length and all the keys of sequence S_i are smaller than all the keys of sequence S_j if $i < j$.

Each processor takes a sample of size $s = \frac{1}{64}n$ of its own n keys at random and sends this sample to one selected processor. This selected processor sorts these $p \cdot s$ sample elements, choose every s-th element of the sorted samples as an splitter element (this gives us splitter elements $s_0, \ldots, s_{p-1}$), and broadcasts them to all processors. Then each processor can distribute its own keys to the p processors in such a way that the key x_j is in sequence S_k if $s_{k-1} < x_j \leq s_k$. In a last step processor i has to sort the sequence S_i. The algorithm is described in detail in [9]. The local sorting of the keys on each processor is done by "radix sort".

If the entire sample size is greater than $32 \cdot p$ (as it is the case for our choice of s) then with high probability the data is split into pieces no larger than $2n$. Therefore, we use one-dimensional arrays A_i of size $2n$ to store the keys which are evenly distributed over the global variables at the beginning.

```
Program of application processor $i$:
splitter phase:
read $n$ keys from the global array $A_i$   (1)
choose $s$ keys at random
write the $s$ samples into the global array $B_i$   (2)
synchronize all application processors
if $i = 0$ then
    read the $p$ sample sequences of size $s$ from the global arrays $B_0, \ldots, B_{p-1}$   (3)
    sort the $p \cdot s$ elements and choose every $s$-th element
    write the $p$ splitter elements into the global array $B_0$   (4)

distribution phase:
synchronize all application processors
if $i \neq 0$ then
    read the $p$ splitter elements $s_0, \ldots, s_{p-1}$ from the global array $B_0$   (5)
    write element $x_j$ into the global array $A_k$ if $s_{k-1} < x_j \leq s_k$   (6)

local sort:
synchronize all application processors
read the keys from the global array $A_i$   (7)
sort the keys
write the sorted keys into the global array $A_i$   (8)
```

To overcome distributing the keys one by one to the application processors in step (6) we split this step into three parts. First, each application processor i writes its own keys into p local lists $L_0, \ldots, L_{p-1}$. Then it writes the lengths of these lists into a global array stored in the shared memory module i (6a). After a synchronization of all application processors each application processor reads the length of every list of every other application processor (6b).

This enables each application processor to compute the memory allocation where it has to store each of its own lists. Then each list L_i is written into the global array A_i at the right destination (6c). That means, application processor i stores its local lists L_j in the global array A_j, $0 \leq j \leq p-1$, at the right destination.

Let l be the packet size and k the send hot spot. Each application processor potentially has to sort a different number of keys, so let $E(s,n)$ denote the expansion factor, i.e., the ratio of the maximum number of keys to sort by a single application processor to n. Thus, $E(s,n) \cdot n$ is the number of elements one application processor has to sort at most. Then the communication time of the single steps can be estimated by the following terms.

$$\begin{array}{llll}
T_{(1)}: & \lceil \frac{n}{k \cdot l} \rceil \cdot T_{k\text{-}k}(k,l,r=1) & T_{(6a)}: & T_{k\text{-}k}(k=1,l=p,r=1) \\
T_{(2)}: & T_{k\text{-}k}(k=1,l=s,r=1) & T_{(6b)}: & T_{k\text{-}k}(k=p,l=p,r=1) \\
T_{(3)}: & T_{k\text{-}k}(k=p,l=s,r=1/p) & T_{(6c)}: & \lceil \frac{n}{k \cdot l} \rceil \cdot T_{k\text{-}k}(k,l,r=1) \\
T_{(4)}: & T_{k\text{-}k}(k=1,l=p,r=1/p) & T_{(7)}: & \left\lceil \frac{E(s,n) \cdot n}{k \cdot l} \right\rceil \cdot T_{k\text{-}k}(k,l,r=1) \\
T_{(5)}: & T_{k\text{-}k}(k=1,l=p,r=1) & T_{(8)}: & \left\lceil \frac{E(s,n) \cdot n}{k \cdot l} \right\rceil \cdot T_{k\text{-}k}(k,l,r=1)
\end{array}$$

The total communication time of the algorithm is the sum of the single communication times.

6.5 Valuation of the results

The ascertained differences between measured and predicted runtimes of k-k-accesses are listed in Table 2. We define the error as $\frac{T_{\text{theor}} - T_{\text{exp}}}{T_{\text{exp}}}$, where T_{theor} denotes the predicted and T_{exp} the measured communication time. One can see that the error is less than 10% if indirect service networks (like BF) are used. The inaccuracy is much greater if direct service networks are used. The reasons for this are given below.

Network	Connected components		Bitonic sort
	T_{bfs}	T_{merge}	T_{bitonic}
SE(3)	321.91 %	60.93 %	57.27 %
SE(6)	627.52 %	208.27 %	13.70 %
CCC(4)	545.42 %	178.42 %	16.44 %
BF(3)	9.15 %	3.57 %	2.42 %

Network	Matrix Multiplication		Sample Sort		
	without sync.	with sync.	splitter phase	distribution phase	local sort
SE(3)	26.44 %	17.73 %	269.13 %	72.15 %	515.96 %
SE(6)	51.71 %	31.62 %	190.01 %	63.61 %	1093.8 %
CCC(4)	67.91 %	51.05 %	159.13 %	60.39 %	934.60 %
BF(3)	0.786 %	3.75 %	10.28 %	79.44 %	32.49 %

Table 2: Differences between measured and predicted runtimes of a k-k-access.

The influence of synchrony on the precision of the model. The precision of the model is good if the application processors are highly synchronized. If the application processors are synchronized before they access some data from global variables then all the application processors access the shared memory almost at the same time and the value of parameter r can be estimated precisely by $r = 1$. If they are not synchronized some application processors still do some communication while others do not because they have already finished or not yet started their communication. So the contention is smaller in the first part and in the last part of the routing, which means $r < 1$, but there is no hope to determine the right value of r, so this leads to an error in the predicted runtime (cf. Table 2, Matrix Multiplication).

The influence of data locality on the precision of the model. Indirect service networks are described very precisely by our model. Each application processor is equally distant from all the shared memory modules and thus there is no notion of locality. The enormous differences in the case of modeling direct service networks can be explained by the lack of modeling data locality. If each access to the global memory is addressed to the service processor nearby there is just one link to cross for the packets and there is no contention in the network. Our model assumes that the locations of shared memory accesses are distributed evenly over all service processors. So the predicted communication times relate to average paths lengths. In the case of k-accesses where each application processor is allowed to access every shared memory module we get an average path length of

$$H_k = \frac{1}{p^2} \sum_{i,j} d(i,j)$$

where p denotes the number of application processors, and $d(i,j)$ the distance between application processor i and shared memory module j. In the case of k-k-accesses where for any $f \in F_k$ and fixed j_0 the function $f(i,j_0)$ to be routed is a permutation, so that the destinations of the accesses are distinct, we get an average path length of

$$H_{k\text{-}k} = \frac{1}{p!} \sum_{i=1}^{p!} \max_{1 \le j \le p} \{d(j, \pi_i(j))\}$$

where π_i denotes a permutation $\pi_i : [1,p] \longrightarrow [1,p]$ and $\pi_i \neq \pi_j$ if $i \neq j$. Applied to indirect service networks both values are the same because the share memory modules are equally distant from the processors, so $d(i,j)$ and $d(j,\pi_i(j))$ are equal to the diameter of the network for each i and j. Both cases imply network contention and a distance greater than 1 for the packets to travel. For example, the average path length for a Shuffle-Exchange service network of dimension 3 in the case of k-accesses is ≈ 2.1, while it is ≈ 3.7 in the case of k-k-accesses. If we take into account some further inaccuracy because of the neglected network contention we expect that the predicted access time is up to 4 times the measured communication time in the case of k-k-accesses (cf. Table 2, row SE(3), column T_{bfs}). This fact shows that modeling data locality is essential for a good prediction of access times in the case that direct service networks are used.

The influence of communication patterns on the precision of the model. The differences between predicted and measured communication times are small if the destinations of the accesses to global variables are evenly spread over the shared memory modules. This is another case where our model describes the access time in direct service networks very precisely. If the amount of accessed data is not equally large for each application processor, as it is the case in the distribution phase of "sample sort", this leads to errors in the prediction of communication times because our model does not consider irregular communications.

Discussion. Our model tries to close the gap between the synchrony of the BSP model and the asynchrony of the LogP model; the asynchrony is large in the case that the send hot spot k is large. Other situations with two further parameters for more general hot spots (in the case of k-accesses) and asynchrony are investigated. In many cases our model is well suited to predict communication times precisely, it is easy to handle, and many situations in parallel programming can be described. Our cost model includes essential parameters in the right combination and neglect non-essential parameters. In the next section we support this by deducing rules for the efficient use of the service network with the help of our model and we validate these rules in practice.

One drawback of our model is the difficulty to determine the right value of r if the application processors are not synchronized which results in inaccurate prediction. Another drawback is that irregular communication patterns can not be modeled. But the main drawback is the lack of modeling data locality.

7 Minimizing the routing time with use of the cost model

As we mentioned in Section 2 the choice of the number and size of packets and the send hot spot has a great influence on the communication time.

We now determine the optimal size k_0 of send hot spot and size l_0 of packets in order to access a fixed amount of data $D = k_0 l_0$ (e.g., a global variable of size 10000 integers) from shared memory modules with minimal communication time. All application processor accesses the same amount of data from distinct shared memory modules, therefore, the value of parameter r is one.

To determine the optimal packet size l_0 and the size k_0 of send hot spots it is necessary to determine the minimum of the trilinear function $T_{k\text{-}k}(k, l, r)$. In the case of $r = 1$ the latency for a k-k-access is described by $T_{k\text{-}k}(k, l, r) = a_3 \cdot kl + a_2 \cdot l + a_1 \cdot k + a_0$ for some $a_3, \ldots, a_0$ (to obtain from Table 1). The derivation of the function $T_{k\text{-}k}(k = \frac{D}{l}, l, 1)$ in order to determine the minimum yields the optimum at:

$$k_0 = \sqrt{\frac{a_2}{a_1} \cdot D} \quad , \quad l_0 = \sqrt{\frac{a_1}{a_2} \cdot D}$$

In fact, this is a rule for the efficient use of service networks. Each time a processor wants to access D items from a global variable the access has to split into k_0 accesses which produces packets of size l_0 each.

As can be seen in Figure 13 the precision of the prediction is quite good. The existence of the optimum and the form of the curve are predicted well. As we expect, the graphs of Figure 3 (page 8) and Figure 13 are very similar.

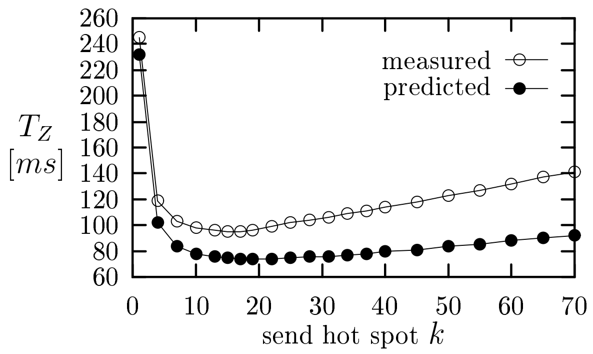
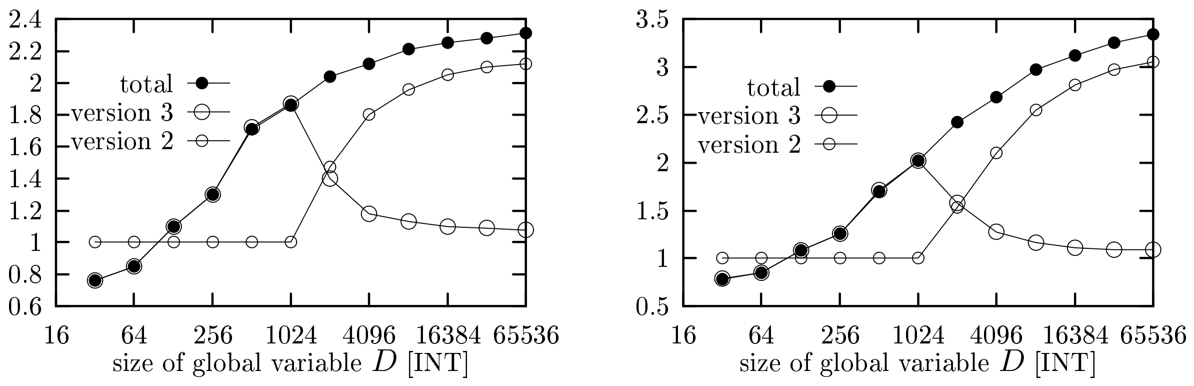


Figure 13: Send hot spot and the packet size influence the communication time (BF(3), $D = 10000$).

Optimization of the routing. The amount D of data which has to be accessed from the shared memory modules is given by the application program (e.g., size of global variables). In general the programmer chooses parameters r, k, and l by himself (as we did in our algorithms above), thus $D = kl$. This means that the router sends successively k packets each of size l in order to access the global variable.

We extend the implemented routing in that way that the router determines values for l_0 and k_0 on-line in the case that all processors accesses global variables ($r = 1$). The optimal values of the packet size l_0 and of the size k_0 of send hot spots depend on the amount of data D, on the kind of network used, and on the number of accessing processors.



(a) SE(6). (b) SE(3).

Figure 14: Achieved speed ups for Shuffle-Exchange networks.

Before doing this in a first optimization step we split each access to global variables into packets of size 4 kByte each and send them one after the other to their destination. Figure 14 (graph "version 2") shows the speed up yielded by this optimization compared with the case of accessing the items in one access.

In a second optimization step the router determines the optimal values for l_0 and k_0. Figure 14 (graph "version 3") shows the speed up yielded by this optimization compared with the case of sending packets of size 4 kByte. The graph "total" shows the achieved speed up by this optimization compared with the case of accessing the items in one access ($k = 1, l = D$).

Because of the complicated on-line computation of the right packet size the communication time for short packets increases. If the size of the variable is greater than 128 integer, the access time decreases significantly. If the size is greater than 16384 integer the communication time can not be reduced any further because the pipelining effect is exhausted, the network is filled with packets. The optimal packet size of k-accesses or in the case $r < 1$ can be determined in the same way.

8 Conclusions

In this report we proposed a new cost model that allows precise predictions of communication times on parallel computers consisting of processors and special purpose routing hardware. It was shown that all important parameters, such as network contention and packet size, are taken into account. These parameters are described in such a way that the analysis of programs is feasible. So the model strikes a balance between detail and simplicity.

We motivated and validated all parameters of the model by showing their influences on the communication time. It turns out that the packet size has a great influence on the efficiency of routing. So we determined the optimal packet size by deriving the trilinear function delivered by our model. To further improve our model one can add another multilinear function that describes the behavior of parallel computers if data locality can be used.

The interaction between asynchrony and receive hot spots is described in detail. We showed that in most situations receive hot spots do not influence the communication time of programs in a high degree.

Acknowledgments

We would like to thank Friedhelm Meyer auf der Heide, Willy - Bernhard Strothmann, and Rolf Wanka for fruitful discussions.

References

[1] A. A., M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 95 – 105, 1995.

[2] A. Aggarwal, A. M. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, pages 3 – 28, March 1990.

[3] A. Aggarwal, A.M. Chandra, and M. Snir. On communication latency in PRAM computation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11 – 21, 1989.

[4] F. Meyer auf der Heide and B. Vöcking. A packet routing protocol for arbitrary networks. *Proceedings of the 12th STACS*, pages 291 – 302, 1995.

[5] K. Batcher. Sorting networks and their applications. *Proceedings of the AFIPS Spring Joint Computing Conference*, 1986.

[6] Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the 3rd European Symposium on Algorithms (ESA)*, 1995.

[7] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 3–16, July 1991.

[8] L. Bomans, D. Roose, and R. Hempel. The Argonne/GMD macros in FORTRAN for portable parallel programming and their implementation on the Intel iPSC/2. *Parallel Computing*, (15):119 – 132, 1990.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subromanian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–11, 1993. Also appears as TR No. UCB/CS/92 713.

[10] M. Fischer and J. Rethmann. *Entwicklung und experimentelle Analyse eines parametrisierten Rechenmodells zur Laufzeitvorhersage paralleler Algorithmen.* Diplomarbeit Universität Paderborn, 1994.

[11] R. Funke, F. Lücking, R. Lüling, B. Monien, and H. Blanke-Bohne. An optimized reconfigurable architecture for transputer networks. In *Proceedings of the 25th Hawaii International Conference on System Sciences (HICSS)*, volume 1, pages 237 – 245, 1992.

[12] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In *6th Annual Distributed-Memory Computer Conference*, pages 258 – 261, 1991.

[13] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. Tr-10-92, Aiken Computation Laboratory Harvard University, 1992.

[14] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 1 – 18, 1992.

[15] P. B. Gibbons. A more practical pram model. In *Proceedings of the ACM Symposium on Parallel Algorithms*, pages 158 – 168, 1989.

[16] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the Twenty-Fourth Annual ACM Symposium of the Theory of Computing*, pages 318–326, May 1992.

[17] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, 1992.

[18] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, pages 339–374, 1984.

[19] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, May 1994.

[20] R. Miller. A library for bulk-bynchronous parallel programming. In *Proceedings of the British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, 1993.

[21] L. G. Valiant. A bridging model for parallel computing. *Communications of the Association for Computing Machinery*, 33:103–111, 1990.

[22] A. Wachsmann. *Eine Bibliothek von Basisdiensten für Parallelrechner: Routing, Synchronisation, gemeinsamer Speicher.* Dissertation, Universität-GH Paderborn, 1995.

[23] A. Wachsmann and F. Wichmann. OCCAM-light – A multiparadigm programming language for transputer networks. Reihe Forschergruppe 5, Universität-GH Paderborn, April 1993.

[24] J. Woo and S. Sahni. Hypercube computing: Connected components. *The Journal of Supercomputing*, 3(3):408–417, 1989.