Latency-driven programming of computer networks

Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Volker Strumpen

Latency-driven Programming of Computer Networks

April 1995

ETH Zürich
Departement Informatik
Institut für Wissenschaftliches Rechnen
Prof. Dr. W. Gander

Volker Strumpen

This report is also available via anonymous ftp from ftp.inf.ethz.ch
as doc/tech-reports/1995/232.ps.

Latency-driven Programming of Computer Networks

Volker Strumpen

April 20, 1995

Abstract

Two properties of computer networks impose major constraints on distributed parallel programming: communication latencies, and permanently changing loads. We propose a technique that faces these problems by combining asynchronous message passing for latency hiding and message continuations for message-driven scheduling of threads. The key to these mechanisms is a portable multithreaded runtime system that can be interpreted as an extension of the TCP/IP protocol suite. This *runtime layer* utilizes idle times of the underlying protocol layers. Experiments show that up to 80 percent of useful computational power can be squeezed out of the CPU while communicating with TCP/IP via an Ethernet and almost 90 percent while communicating across the Internet. First-class message continuations are provided to cope with desynchronization and message delays. A message continuation is executed after a message is sent or received. This dependency on communication latency coined the name *latency-driven* programming model.

1 Introduction

Computer networks are becoming a primary compute resource for engineers and scientists. Heterogeneous computing on networks of supercomputers is being promoted for tackling very large problems, whereas technological advances and the wide-spread availability of workstation networks make these an attractive, inexpensive alternative for daily use. Although CPU speed and memory capacity of workstations exceed supercomputer node performance, interconnect technology limits the use of clusters for parallel computing to coarse grained applications. Furthermore, clusters are shared by users, and permanent unpredictable changes of CPU load and network load complicate the efficient utilization of clusters for distributed parallel computing.

It is apparent from technological trends that the current bottleneck of computer networks, the network, is shifting towards the main memory module due to the introduction of high performance networks and fast host-network interfaces. Current research, including the work described here, is emerging to level the difference between multiprocessors and workstation networks, and that of local and remote data access. Thus, memory latency hiding techniques are gaining importance also for local data access. Whereas the work on communication latency hiding described here focusses on utilizing protocol idle times of machines with DMA-based host-network interfaces [22], it is expected that future systems make less use of the CPU for protocol processing. With such system architectures, as for example developed in the *T project [17] or HPAM [13] based on the Medusa host-network interface [1], even more efficient latency hiding implementations will be possible.

To overcome the difficulties of communication latency and load changes in computer networks, a programming model is proposed that:

1. Supports latency hiding by means of asynchronous send and receive operations. Here, the attribute *asynchronous* denotes the separation of a send or receive operation into a non-blocking registration operation and a corresponding synchronization operation.

2. Avoids message copying by synchronizing communication threads in favor of utilizing system buffers.

3. Copes with unpredictable synchronization and message delays by means of *message continuations*. These are pointers to functions that are executed locally as soon as the corresponding messages are sent or received.

The combination of latency hiding and message-driven execution is the key aspect of this design. The message continuation can be viewed as an orthogonal construct to an active message [24]: Whereas the active message contains an address of the handler that is to be executed on the receiver side, the message continuation is registered and executed at the registration side as soon as a message transfer is completed, i.e. the message has either left user space or has been received in user space. Thereby, compile time scheduling decisions for latency hiding can adapt to the actual runtime of message transfer: If not all instructions scheduled for latency hiding are executed during the transfer, the message continuation can initialize another transfer to utilize those instructions.

A runtime system has been implemented that is portable across UNIX operating systems and utilizes TCP/IP idle times for latency hiding. The runtime system is multithreaded, distinguishing a communication thread, user-defined message handlers and message continuations that are executed as individual threads. Furthermore, one or multiple application threads can be used. In contrast to most multithreaded runtime systems that are provided just for the sake of first class user-level threads [10, 7, 15], this implementation is primarily based on multithreading in order to schedule communication and application threads. The multithreading support needed is just a context switch. Therefore, the implementation is based on the QuickThreads package [11], which provides this basic functionality in a portable manner. The runtime system is linked to an application written in the C language. Support for process and communication link management is not offered in the runtime layer, and must be provided by systems such as PARC [20].

This work focusses on clusters distributed over large geographical distances. The latency of long-haul networks connecting these clusters is not expected to decrease by several orders of magnitude in the near future. Network properties such as relatively high error rates and the need for flow control are well handled by protocols such as TCP. The presented experiments show that efficient latency hiding can be provided if these networks approach Ethernet throughput rates of 10 Mbit/s. For local clusters with workstations at close proximity to each other, future high performance networks offer the chance to decrease latency of short messages by at least an order of magnitude. This requires major changes of the communication path in the operating system. Faster mechanisms than system calls, for example traps, might be utilized. However, even if additional communication processors become available, the latency-driven programming model remains valuable although its implementation will change.

2 Communication Latency Hiding

The rationale of latency-driven network programming can be grasped with the help of the following model of communication latency hiding. For a detailed treatment the reader is referred to [21, 22]. Consider a domain decomposition application, which comprises a loop that contains a calculation and a subsequent communication phase. Typically, the loop iteration models some time dependency. Within each iteration new data are calculated, and then exchanged among cooperating tasks. With the runtime t_{seq} of one loop iteration of the sequential program and equal distribution of work, each task of the parallelized program with p processors is assumed to require calculation time $t_{calc} = t_{seq}/p$ plus communication time t_{com} for one iteration. Now, if only the fraction $(1-f), 0 \leq f \leq 1$, of the calculation time is required to calculate those data that are to be sent to another task, the loop body can be restructured to allow for overlapping communication and calculation. First, the data to be communicated are calculated in time $(1-f)t_{calc}$. Then, these data are registered with the communication layer. Next, the remainder of the task is calculated in time ft_{calc}. Finally, the pending messages are received. Since time ft_{calc} is that portion of calculation time available for overlapping communication, the factor f is called *latency hiding degree*.

With these assumptions the runtimes of the parallel implementations without latency hiding t and with latency hiding t_{lh} can be approximated by:

$$\begin{aligned} t(p) &= t_{calc} + t_{com}, \\ t_{lh}(p) &= (1-f)t_{calc} + \max\bigl(ft_{calc}, t_{com}\bigr). \end{aligned}$$

Communication latency hiding is characterized by means of *gain* G, which is defined here as the ratio of the speedups S_{lh} of the implementation with latency hiding and S without latency hiding:

$$\begin{aligned} S(p) &= \frac{p}{1 + t_{com}/t_{calc}}, \\ S_{lh}(p) &= \frac{p}{(1-f) + \max\left(f, t_{com}/t_{calc}\right)}. \end{aligned}$$

Introducing the *software granularity* $\gamma = t_{calc}/t_{com}$, we obtain gain

$$G = \frac{S_{lh}(p)}{S(p)} = \frac{1 + 1/\gamma}{(1-f) + \max(f, 1/\gamma)}.$$

In this representation, $f = 0$ corresponds to the case without latency hiding, and $f = 1$ to *ideal* latency hiding. Figure 1 illustrates speedup and Fig. 2 the corresponding gain for varying latency hiding degree f.

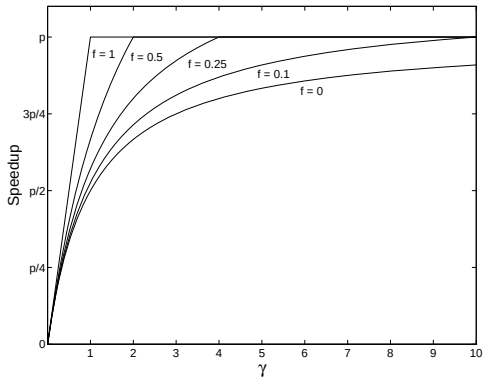


Figure 1: Dependency of speedup $S_{lh}(p)$ on communication latency hiding degree f and software granularity γ

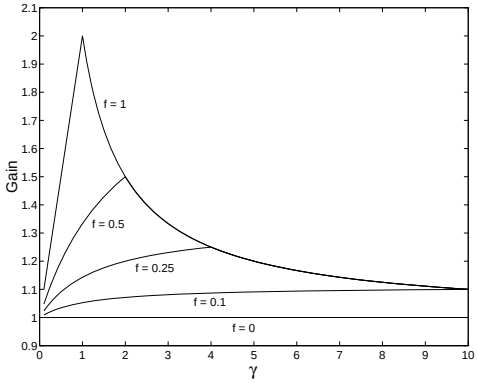


Figure 2: Gain of communication latency hiding

Gain G is bounded by the constant 2. Furthermore, G is directly related to efficiency. With the definition of efficiency $E(p) = S(p)/p$, gain equals the quotient of efficiencies of the versions with and without latency hiding: $G = E_{lh}/E$. It should be pointed out that, using single-threaded processes, without a high value of gain (i.e., close to 2), utilizing larger numbers of processors will in general lead to low efficiency for relatively large communication times. Therefore, gain itself is

a secondary objective which will result in higher efficiency, and leads to a solution that permits the use of a particular number of processors with maximum efficiency or speedup.

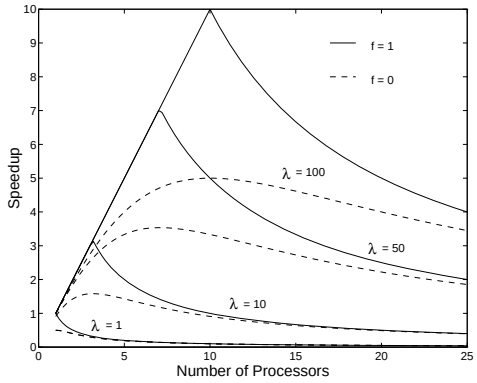


Figure 3: Speedup dependency on granularity λ in a multiprocessor system with ideal and without latency hiding.

In the previous discussion, the dependency on the number of processors has been hidden in the software granularity term $\gamma = t_{calc}/t_{com}$. Now, p becomes an explicit parameter of gain G, when substituting granularity $\lambda := p\,\gamma = t_{seq}/t_{com}$ for γ. Figure 3 illustrates the dependency of speedup on λ with respect to the number of processors. The family of dashed curves corresponds to the case without latency hiding ($f = 0$), the solid lines to the case of ideal latency hiding ($f = 1$). The dashed speedup curves attain a global maximum, which is typical for applications whose communication volume is proportional to the number of processors employed. Speedup decreases beyond this maximum, because communication overhead outweighs calculation time. The maximal values of speedup with ideal latency hiding are achieved with the number of processors yielding also the maximal gain. For decreasing λ, the number of processors operating with maximal speedup and efficiency decreases. This coincides with the general notion that software granularity, hardware granularity, speedup, and efficiency are directly related to each other. Furthermore, it is obvious that latency hiding is a means of increasing scalability.

The conclusion of this analysis is the objective to maximize gain G. The challenge is to schedule instructions in a way that communication is hidden entirely, i.e. globally across all communications. This is an NP-hard problem already for dedicated machines with predictable communication delays and static compile time analysis. In computer networks with varying resource loads dynamic mechanisms are necessary to approach the peak of the gain curve.

3 Programming Model

It is not the intention of the currently implemented programming model to provide a complete syntax or elegant abstractions. Instead, functions are used as a library extension of the C language to access the runtime layer functionalities. The programming model is built upon the conventional point-to-point message passing model. The basic primitives are non-blocking send and receive operations to register the message with the runtime layer, and the corresponding synchronization routine that blocks a thread until the requested transfer is finished. This interface is similar to the `isend/irecv` and `msgwait` routines of Intel's NX message passing interface [18]. The (ANSI C) syntax of the asynchronous message passing operations is as follows:

```
int send(int pid, int sno, void (*mf)(margv_t *), margv_t *ma)
int recv(int pid, int sno, void (*mf)(margv_t *), margv_t *ma)
int msync(int mid)
```

Both the `send` and `recv` operation return a unique message identifier, which is used as the argument for an `msync` call. `msync` possibly returns error codes indicating communication failures or broken connections. All statements between a `send` or `recv` and the corresponding `msync` are available for hiding the communication initiated by the `send` or `recv` call. In the following this set of statements is called *overlap region*. The argument lists of `send` and `recv` are identical: `pid` denotes the destination process identifier, `sno` a sequence number that allows the user to impose an order on message delivery, `mf` is a pointer to a marshal function, and `ma` a pointer to the argument list of type `margv_t` for `mf`. `mf` and `ma` form the closure for a marshal thread which executes the function `mf(ma)`.

The marshal thread is a program structuring mechanism, but in the first place introduced to increase the performance of a single message transfer by pipelining marshaling and message transfer. If an array consisting of basic data type elements can be transferred without conversion of the data representation, a marshal function is not needed. If marshaling is required, each message is uniquely associated with a marshal thread. Within the marshal routine, basic data types are written into or read out of a stream, which is mapped to message fragments within the runtime layer. Usually, the arguments passed to `mf` via `ma` are linearized within `mf` and written to or read from the stream. If the marshal thread terminates, the end of the message is determined by the runtime layer. This way, the programmmer can transfer a message without knowing its actual length. Of course, the marshal function and the corresponding unmarshal function must match. A set of macros is used within the marshal functions to linearize, and convert the data representation in heterogeneous environments if necessary. Besides this original purpose of the marshal thread, it can be used to spawn local threads or provide other interesting abstractions.

The `send` and `recv` operations start the transmission of a message. Given the number of statements in the overlap region, a fixed amount of time, which depends on the CPU load, is reserved for this calculation. If the message transmission takes less time than this period, the remaining statements cannot be used for latency hiding. For this situation message-driven control is provided in terms of *message continuations*. A message continuation is a pointer to a function that is executed if a predefined message event happens. The continuation and the corresponding message event are defined with the following routine:

```
int mcont(int mid, void (*cf)(margv_t *), margv_t *ca)
```

Since the scope of a message identifier is limited to the overlap region, `mcont` must be invoked within this overlap region. This call registers the message continuation with message `mid`. The continuation `cf` and the pointer to its argument structure `ca` compose a closure that is passed to a thread for execution of the message continuation. Two events can trigger the execution of the message continuation: If `mid` identifies a message that is to be sent, the continuation is started as soon as the message has left the runtime layer. If `mid` denotes a message that is to be received, the continuation is started as soon as the message arrives in user space. Thus, the dynamically changing communication latency determines the start of execution of a message continuation. Because of this mechanism, we call the emerging style of programming *latency-driven*.

The concept of message continuations covers those applications that can provide a sufficient amount of work for overlapping communication. If this cannot be guaranteed, the programmer might employ conventional multithreading. The major advantage of message continuations versus multithreading is the immediate action in case of message arrival as well as departure.

Besides the operations described above, the runtime layer supports asynchronous active messages. Overlapping can be used to hide the communication latency of large active messages. Other mechanims for message control and thread synchronization are currently under investigation.

4 Internal Design Issues

In this section, those implementation ideas are described that assist in understanding the semantics of the programming model. For the sake of clarity, details are omitted.

Our interest focusses on applications with large resource requirements with respect to computational power, memory and communication volume as needed for parallel scientific computations,

for example. Furthermore, portability is a design constraint that is indispensible in order to work with large networks of machines [20]. The key component enhancing state-of-the-art systems is a runtime layer that extends the TCP/IP protocol suite [19] where it is sandwiched between the transport layer and the application layer. The runtime layer integrates multithreading and a protocol for multiplexing and demultiplexing message transfer by means of non-blocking `read` and `write` system calls to the Berkeley socket interface. Although these calls are rather expensive, this decision was made for the sake of portablity. Furthermore, for large messages the overhead of these calls is tolerable.

4.1 Buffering and Synchronization

It is anticipated that primary memory is becoming the bottleneck of future high-performance networking [23]. To this end it is mandatory to avoid unnecessary copies (buffering) of large messages in order to ameliorate the bottleneck of memory bandwidth. For short messages protocol overhead dominates the so-called startup time. Here, message buffering overhead is negligible. The difference of the mechanisms limiting the transfer performance led to the following design decision for the runtime layer protocol:

Small messages ($\leq$ 4KByte) are transferred immediately and buffered in the receiver's runtime layer, if the application is not yet ready to receive the message. Large messages ($>$ 4 KByte) are only transferred after the receiver is ready to copy the message immediately into user space.

Measurements show that the 4 KByte tradeoff is valid across the range of current computer networks [21]. Synchronizing the communication partners of large messages is based on fragmentation. Large messages are fragmented into pieces of 4 KByte. If the sender arrives at the `send` statement prior to the receiver at the `recv` statement it submits the first fragment which is stored in the receiver's runtime layer. As soon as the receiver posts the corresponding receive operation, a (short) request message is sent to the sender, which resumes to transmit the message fragmentwise. These fragments are not buffered in the runtime layer but directly written into the receiver's user space. If the receiver arrives at the `recv` statement before the first fragment of the message arrived, it sends the request immediately. If the sender starts transmission, the whole message can be copied directly into the receiver's user space. On the sender side, messages are only buffered in the runtime layer for marshaling purposes. Synchronization of sender and receiver might delay a thread's execution if it is blocked in a `msync` operation. However, in a properly structured application, multiple threads will be available that can be scheduled to utilize the idle CPU in the meantime. This (pretty complicated) protocol offers two major advantages:

- The overall copying overhead in a parallel program is reduced.
- The implementation is space efficient by using bounded buffers of the size of a fragment (4KByte) in the runtime layer. Furthermore, it provides deadlock freedom which is not guaranteed by the underlying transport layer (TCP).

The runtime layer assumes a reliable transport layer with FIFO-ordered message delivery. The current implementation is built on top of TCP, because it accomodates these properties. Nevertheless, faster networks and underlying protocols should make the advantages of this concept even more obvious.

4.2 Overlapping Calculation and Communication

The implementation of communication latency hiding is based on the distinction of application threads, marshal threads and a single communication thread. The communication thread multiplexes outgoing messages and demultiplexes incoming messages of the entire application process. Optionally, each message is associated with a unique marshal thread. An application thread can handle various marshal threads simultaneously.

The communication thread manages message transfer by means of hashed and orthogonally linked lists of data structures containing the message states. A message identifier, carried by

the message is used to look up the corresponding data structure on receipt by means of hashing. Lists are used to gather all messages of a process with respect to their states. With these data structures, the communication thread is able to multiplex all messages of a process across all virtual connections independent of the state of individual application threads. Communication is based on non-blocking system calls. If a message fragment cannot be launched into or pulled out of the kernel after a predefined number of retries, the multiplexer switches to another socket. If none of the sockets is ready after a predefined number of sweeps across all sockets, the communication thread switches back to an application thread.

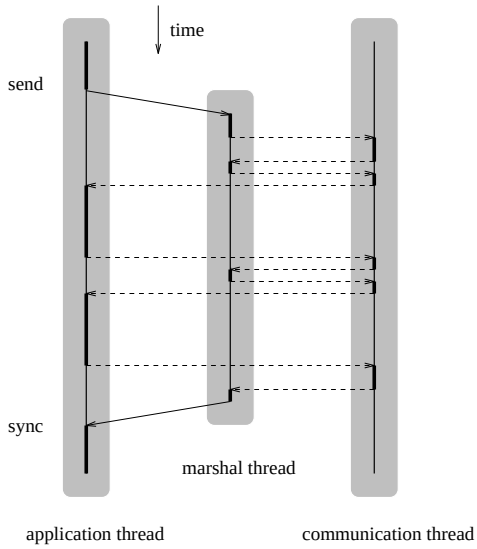


Figure 4: Interleaved message handling in one process consisting of three threads: An *application thread* that sends a message using the message-specific *marshal thread*, and a process-specific *communication thread* that interfaces to TCP/IP. Broken lines indicate context switches, and bold lines mark the running thread.

The overlapping of calculation and communication is implemented by interleaving application threads, marshal threads and the communication thread. By integrating the communication thread into the thread scheduling policy superfluous context switches are avoided, and a sensible polling of the network is possible. If messages are pending, the communication thread is preemptively scheduled after a predefined timeout. In the current implementation this timeout equals the resolution time of the alarm timer, on SPARCstations $20\,ms$.

Figure 4 illustrates the interleaving of an application thread, a marshal thread and the communication thread when sending a message. During the send operation, a marshal thread is spawned. After a fragment has been marshaled into the bounded runtime buffer, a non-preemptive context switch is performed to the communication thread. If the fragment has been delivered successfully to the local runtime layer, the communication thread returns to the marshal thread to pack the next fragment. The communication thread switches back to the application thread, if the transport layer refuses to accept any further bytes. If the message is still pending, the context preemptively switches back to the communication thread after the timeout. The network can recover or transfer messages of other processes while the application thread is running. When the entire message is consumed by the runtime layer, the marshal thread is terminated and control is returned to the application thread. If a message continuation had been registered, execution would be started at that moment.

To reduce the number of unsuccessful `read` and `write` system calls, continuations [6] have been used within the runtime layer to avoid busy waiting during the read or write of a single message fragment. This optimization greatly increased the efficiency of latency hiding. A detailed treatment of the scheduling policy is beyond the scope of this paper and appears elsewhere.

5 Experimental Results

In order to measure the latency hiding capabilities of the runtime layer a synthetic benchmark has been created. Two processes simultaneously exchange two messages of equal size. The runtime layer ensures that this is possible without deadlock. We measure the amount of useful computation that is done within the overlap region. The BLAS `daxpy` ($\vec{y} = a\vec{x} + \vec{y}$) [12] has been chosen as a well known computational kernel to benchmark the CPU power available for useful computation. It is applied to double precision vectors of length 1000. A message continuation is used to terminate an infinite loop of daxpy computations. The resulting count of daxpy executions is used as the measure of useful CPU utilization overlapping the communication.

The following code fragment contains the kernel of this benchmark:

```
int flag;

void contf(void)
{
   flag = 0;
}

void
bench(int pid,margv_t *sma,margv_t *rma)
{
   int smid, rmid, dc=0;

   rmid = recv(pid,0,unmarshal,rma);
   smid = send(pid,0,marshal,sma);

   flag = 1;
   mcont(smid,contf,0);

   while (flag)
      fdaxpy(), dc++;
   printf("dc   %d"\n, dc);

   msync(smid);
   msync(rmid);
}
```

A global flag is used to terminate the daxpy loop by switching it within the message continuation function `contf`. The semantic of this flag corresponds to that of a semaphore. In order to reset the flag it must be set before the message continuation is executed. Within the C function `fdaxpy` the original FORTRAN routine is called. Storage for vectors $\vec{x}$ and $\vec{y}$ is allocated elsewhere.

Table 1 and Table 2 list the results of this benchmark for a LAN configuration consisting of two SPARCstation10 connected via Ethernet. Two processes exchange messages simultaneously with payloads varying from 4 KByte to 1 MByte. The listed transfer times correspond to the case without running the daxpy loop and without message continuations. The dedicated daxpy counts are measured with a daxpy loop that is interrupted after a period equal to the transfer time by means of an alarm signal. The send daxpy counts are generated with the benchmark program given above, and the receive daxpy counts with the message continuation parametrized with the receive message (`rmid`).

The difference between Table 1 and Table 2 is the application of marshaling: For the measurements presented in Table 1, an array of size payload is transferred without marshaling (`marshal` = `unmarshal` = `(void *) NULL`). In contrast, the values shown in Table 2 are generated with marshaling, where double arrays of size payload are marshaled by copying each double value into and out of the message. This gives a notion of the overhead for marshaling dynamic data structures. For clarity, no representation conversion is performed.

These measurements demonstrate the viability of the runtime layer approach as an additional layer of the TCP/IP suite in order to improve CPU utilization. Note that the communication times of the experiments with latency hiding are equal to the listed transfer times without overlapping calculation within measurement accuracy. The differences between the send-triggered and receive-triggered versions can be explained with the interleaving of the message fragments on the bus network. Depending on which of the two messages is actually transferred first, these differences vary slightly from measurement to measurement.

It should be noted in this context, that all measurements where obtained in a normal working environment. It was just taken care that no CPU intensive processes were running during the experiments. Furthermore, the bandwidth of the Ethernet is almost optimally utilized. For instance, according to Table 1, two 1 MByte messages are exchanged over the Ethernet within 1.849 seconds, which corresponds to a throughput of 9.1 Mbit/s.

The effectiveness of the latency hiding protocol is the ratio of the daxpy counts executed for latency hiding and the dedicated daxpy counts. As a result, the version without marshaling achieves almost 80 percent useful CPU utilization while communicating (sending and receiving a message). The program with marshaling employs the CPU additionally for copying the array elements into and out of a message buffer. Despite of this loss of power for useful computation, more than 50 percent of the CPU are utilized.

Payload [Kbyte]	Transfer [ms]	dedicated daxpy	daxpy (send)	daxpy (recv)
4	10	40	34	37
8	19	62	56	56
16	31	86	66	64
32	61	153	121	108
64	118	275	199	225
128	233	545	392	413
256	463	1070	784	749
512	931	2140	1444	1522
1024	1849	4200	2978	2992

Table 1: Latency hiding benchmark in a LAN: Two SPARCstation10 connected via Ethernet, whithout marshaling of messages.

Payload [Kbyte]	Transfer [ms]	dedicated daxpy	daxpy (send)	daxpy (recv)
4	12	44	32	31
8	20	62	28	59
16	34	91	58	90
32	66	160	87	111
64	122	297	152	177
128	236	550	293	283
256	461	1070	573	582
512	931	2148	1134	1107
1024	1878	4290	2246	2163

Table 2: Latency hiding benchmark in a LAN: Two SPARCstation10 connected via Ethernet, with marshaling of messages (copying of single double values).

Whereas the variance of the measurements in our Ethernet LAN is small, timings in the Internet are hardly reproducible, and easily vary by a factor of two. Therefore, the latency hiding

potential of the Internet is measured with a single benchmark run, executing a (send) message continuation, and measuring the dedicated daxpy loop count with respect to the transfer times of this program. Table 3 lists the results. Experiments with a receive continuation yielded equivalent values. Marshaling induces only a minor overhead compared to the high latency of the Internet connection.

Payload [Kbyte]	Transfer [ms]	dedicated daxpy	daxpy (send)
4	391	661	410
8	471	717	477
16	1180	1829	637
32	1362	2108	1677
64	4275	7665	5639
128	6021	10860	9158
256	11613	20556	17753
512	24869	40452	35429
1024	39660	71818	61525

Table 3: Latency hiding benchmark in the Internet: One SPARCstation10 at ETH Zürich and one SPARCstation2 at MIT, Cambridge. Send-daxpy counts are measured on the SPARCstation10.

Because the latency of the Internet is roughly one order of magnitude bigger than that of a 10 Mbit/s Ethernet, a higher CPU utilization rate can be expected. In fact, the daxpy counts for large messages achieve rates close to 90 percent. Definitely, these numbers justify an additional layer in the protocol stack.

In order to compare these results of commodity workstation hardware with the latency hiding capability of a specialized parallel machine, we present some experimental data, obtained from an Intel Paragon system. This multicomputer employs an additional communication processor per node in order to enhance the communication capabilities. Intel's NX programming model [18] offers asynchronous communication by means of the `isend/irecv` and `msgwait` routines.

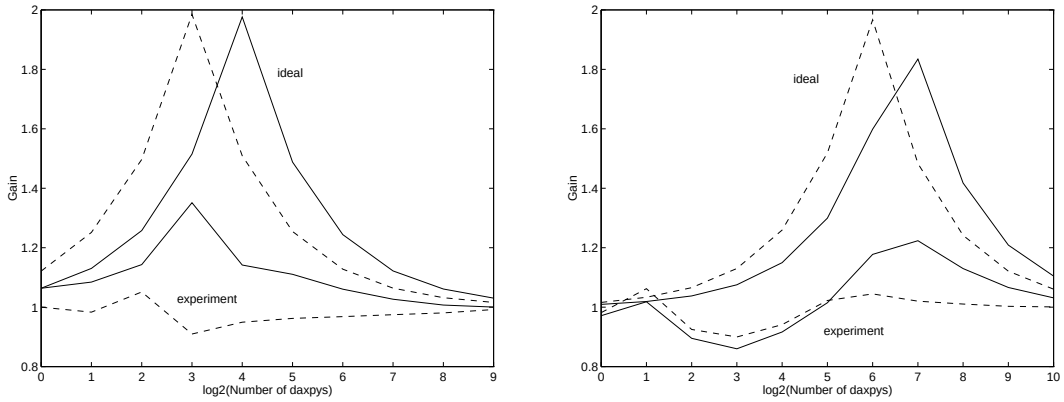
Since no user-level continuations are available to run a benchmark similar to the one above, a different experiment was designed: Consider a fixed communication volume and a varying amount of computation, which is executed between an `irecv` and `isend` call and the corresponding `msgwait` calls in analogy to the benchmark above (simultaneous message exchange). This programs yields the runtime t_{lh} for overlapping communication and calculation. With similar programs we can measure the runtime t_{com} of the dedicated communication, and t_{calc} of the dedicated calculation. With these values, we calculate the ideally achievable gain of latency hiding:

$$G_{ideal} = \frac{t_{calc} + t_{com}}{\max(t_{calc}, t_{com})}$$

and the actually measured gain on the Paragon:

$$G_{exp} = \frac{t_{calc} + t_{com}}{t_{lh}}$$

Figure 5 presents the resulting gain values for two different message sizes: 8 Kbyte and 64 Kbyte. The corresponding communication times are $t_{com}(8\,\text{Kbyte}) = 0.33\,ms$ and $t_{com}(64\,\text{Kbyte}) = 2.5\,ms$. Communication across the Paragon network is approximately one order of magnitude faster than that across a 10 Mbit/s Ethernet as used above. The calculation consists of a loop that accomodates a call to the `daxpy` routine. The amount of calculation is varied by changing the loop iteration count. In order to run measurements in the vicinity of the extreme case $t_{calc} = t_{com}$ we chose the length for the `daxpy` vectors to be 100 double precision numbers. The runtime of a single `daxpy` execution is $40\,\mu s$ (with the `-Mvect -Mstreamall` options). Note that all vectors fit into cache (4 Kbyte) in contrast to the workstation benchmark presented above. Because the memory



(a) Message size: 8 Kbyte

(b) Message size: 64 Kbyte

Figure 5: Gain of communication latency hiding on the Intel Paragon for fixed communication volume and varying amount of computation. The code yielding the solid lines is compiled with option `-O4`, that to obtain the broken lines with `-O4 -Mvect -Mstreamall`.

bus is not accessed in order to supply the processor with the data necessary for the calculation, this favors the Paragon considerably. However, even for this synthetic situation, the gain values are so poor that we do not investigate this aspect further. The measurements present average values of several executions that were preceded by a dummy communication and a dummy `daxpy` execution in order to exclude perturbations due to channel setup and cache misses.

The experimental resuls deviate considerably from the ideal curve. When using the compiler options `-Mvect -Mstreamall`, the values of gain are mostly less than one. This means that overlapping calculation and communication introduces that much overhead that the execution takes even longer than executing communication and calculation one after each other. Since the `-Mvect -Mstreamall` compiler options enable the generation of code that bypasses the cache to fill the pipelines of the processor, memory bus conflicts occur, because both calculation processor and communication processor compete for memory access. This will degrade performance compared to the version where all data is kept in cache (solid lines). The question why the gain of this version is still pretty low is left as an exercise. Concluding, these measurements show that communication latency hiding is very poor on the Paragon and may even be counterproductive. Comparing this result with the communication latency hiding potential of workstations that reaches up to 80-90% with the new protocol is surprising.

6 Related Work

The results presented here outperform those of a previous implementation of a latency hiding protocol of the author described in [21]. That work is based on a two-process setting, one acting as a communication process, the other as calculation process. Messages are exchanged between these processes via shared memory and transferred across processors by means of the communication processes. This scheme is relatively simple to implement. The scheduling task is performed by the original operating system scheduler. The process switches induce a higher overhead than the thread based implementation described here. Both implementations differ with respect to their latency hiding performance. Internet experiments are about 10 percent more efficient with the runtime layer implementation, and Ethernet performance improves from about 60 percent to 80 percent. However, the new runtime layer offers more powerful mechanisms for network computing: multithreaded applications, marshal threads, and message continuations.

Several projects have been emerging recently to ameliorate the problems of traditional message passing style programming. Among these are generic models such as Active Messages [24], and

those aiming at better performance and/or easing parallel programming, by means of message-driven execution models. MDC [14] is a message-driven programming system for the J-machine [5] that introduces a code distribution scheme to cope with the shortage of memory on this machine. More closely related to the latency-driven approach are Charm [9] and Cilk [2].

Active Messages are suited for substantially more efficient implementations than conventional message passing protocols. This model is optimized for handlers executing small amounts of computation, and thus relatively short messages. With respect to the programming model, active messages and the other message-driven systems mentioned here can only act at the instance of message arrival. In contrast, message continuations are provided for large messages, they act locally (on message receipt similar to a UNIX signal), and allow for action on behalf of both message arrival and departure.

Cilk focusses on efficient thread scheduling. Its main contribution is a work-stealing scheduler fulfilling theoretical efficiency criteria that are also matched in practice. The network implementation is based on Phish [3], a predecessor of Cilk. Phish implements split-phase operations based on UDP. This scheme allows for useful computation between a request and the corresponding reply, but not for overlapping communication itself.

Charm is one of the first systems advocating a message-driven programming style. Similar to Cilk, Charm provides automatic mapping and scheduling although with different programming abstractions. So-called chares are programmed by providing entry points for messages. Messages are received by the Chare kernel and buffered in a message queue. A queue manager picks a message according to a predefined load balancing strategy, and starts execution of the code associated with the entry point carried by the message. Message reception and sending use explicit buffering to provide non-blocking communication. Furthermore, the UDP based communication protocol of the networking implementation does not overlap communication with useful computation.

The Distributed Filaments [8] approach applies multithreading for overlapping communication and calculation in a user-level distributed shared memory system. This implementation is based on a split-phase operation to handle a remote page fault. After a request is sent off the local scheduler switches to another thread while waiting for the requested page. Like Charm and Phish, distributed filaments implements a reliable transport protocol on top of UDP, without communication latency hiding as proposed in this paper.

The latency-driven programming model was also stimulated by the work on TAM [4] and P-RISC [16]. In contrast to these models, the work described here is guided by the requirement of portability on top of standard operating system software for large-scale distributed computing.

7 Conclusions

A runtime layer is introduced on top of the transport layer of the TCP/IP protocol suite. This layer utilizes idle times of the underlying protocol layers for useful computation while communicating data. Experiments show that there is sufficient idle time in these layers to overlap communication almost entirely (80–90 percent), if payloads larger than 4 KByte are transferred. Furthermore, message continuations are introduced that react on message arrival or departure, and thereby constitute a mechanism to cope with dynamic communication latencies and desynchronization. It is intended to investigate the suitability of the runtime layer for distributed parallel computing as well as operating system services such as network file systems or distributed virtual memory.

Acknowledgement

Thanks to Peter Arbenz and Rolf Strebel for fruitful discussions and valuable comments, Walter Gander for his encouraging support, and Tom Casavant for focussing my interest on this subject and numerous delighting discussions.

References

[1] D. Banks and M. Prudence. A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202, February 1993.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, A. Shaw, and Y. Zhou. Cilk: An efficient multithreaded runtime system, December 1994. (URL `http://theory.lcs.mit.edu/pub/cilk/cilkpaper.ps.Z`).

[3] R. D. Blumofe and D. S. Park. Scheduling large-scale parallel computations on networks of workstations. In *3rd International Symposium on High-Performance Distributed Computing*, pages 96–105, San Francisco, CA, August 1994.

[4] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.

[5] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, pages 23–39, April 1992.

[6] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *13th ACM Symposium on Operating Systems Principles*, pages 122–136, Pacific Grove, CA, October 1991.

[7] I. Foster, C. Kesselman, and S. Tuecke. Nexus: Runtime support for task-parallel programming languages. Technical report, Argonne National Laboratory, 1994. (URL `ftp://ftp.mcs.anl.gov/pub/nexus/reports/nexus_paper.ps.Z`).

[8] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First USENIX Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, CA, November 1994.

[9] A. Gursoy, S. Krishnan, A. B. Sinha, and L. J. Kale. *The Charm (4.0) Programming Language Manual*. Department of Computer Science, University of Illinois at Urbana Champaign, January 1994. (URL `http://www.cs.uiuc.edu/~rmartin/hotipaper.ps`).

[10] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. Technical Report 94-25, ICASE, April 1994.

[11] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, Department of Computer Science and Engineering, University of Washington, May 1993.

[12] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[13] Richard P. Martin. HPAM: An active message layer for a network of HP workstations. In *Hot Interconnects II*, 1994. (URL `http://http.cs.berkeley.edu/~rmartin/hotipaper.ps`).

[14] D. Maskit and S. Taylor. A message-driven programming system for fine-grain multicomputers. *Software—Practice and Experience*, 24(10):953–980, October 1994.

[15] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX Conference*, pages 29–42, San Diego, CA, January 1993.

[16] R. S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, LNCS 768, pages 390–405, Portland, OR, August 1993.

[17] G. M. Papadopoulos, R. Greiner, and M. J. Beckerle. *T: Integrated building blocks for parallel computing. In *Supercomputing'93*, pages 624–635, November 1993.

[18] P. Pierce. The NX Message Passing Interface. *Parallel Computing*, 20(4):463–480, April 1994.

[19] R. W. Stevens. *TCP/IP Illustrated: The Protocols*. Addison-Wesley, Reading, 1994.

[20] V. Strumpen. A Large-Scale Metacomputer Approach for Distributed Parallel Computing. In *High-Performance Computing and Networking*, LNCS 797, pages 278–285. Springer-Verlag, Munich, April 1994.

[21] V. Strumpen. Communication Latency Hiding — Model and Implementation in High-Latency Computer Networks. Technical Report 216, Department Informatik, ETH Zürich, June 1994. (URL `ftp://ftp.inf.ethz.ch/doc/tech-reports/1994/216.ps.Z`).

[22] V. Strumpen and T. L. Casavant. Exploiting communication latency hiding for parallel network computing: Model and analysis. In *1994 International Conference on Parallel and Distributed Systems*, pages 622–627, Hsinchu, Taiwan, December 1994. IEEE.

[23] A. N. Tantawy, editor. *High Performance Networks. Frontiers and Experience*. Kluwer Academic, 1994.

[24] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.