Communication latency hiding

model and implementation in high-latency computer networks

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Wissenschaftliches Rechnen

Volker Strumpen

Communication Latency Hiding — Model and Implementation in High-Latency Computer Networks

June 1994

216

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

Volker Strumpen
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland
e-mail: strumpen@inf.ethz.ch

Communication Latency Hiding Model and Implementation in High-Latency Computer Networks

Volker Strumpen

Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich

Abstract

The potential of large numbers of workstations for solving very large problems is tremendous. Nevertheless, it is often considered inappropriate to parallelize applications with a fair amount of communication on computer networks, because communication via networks with high latency and low bandwidth presents a technological bottleneck. In this paper, a model to analyze the gain of communication latency hiding by overlapping computation and communication is described. This model captures the limitations and illustrates the opportunities of communication latency hiding for improving speedup and efficiency of parallel computations that can be structured appropriately. Furthermore, an implementation of a message passing protocol is presented that incorporates latency hiding on top of the TCP/IP transport layer. This protocol ensures efficient, deadlock-free communication in UNIX network environments. Experiments show that the presented latency hiding technique increases the range of applications suited for parallel computing on networks of computers, even across the Internet. Measurements with a multiprocessor system demonstrate the validity of the latency hiding model for a broader range of parallel architectures. Parallel programming with conventional message passing interfaces is only slightly affected, because an additional protocol layer hides the increased complexity from the programmer.

1 Introduction

Currently, an increasing acceptance of network computing can be observed among engineers and scientists [2,3,4,13,17]. Network computing is defined to be utilization of some software platform to support distributed parallel computing in a heterogeneous computer network, usually consisting of a large number of workstations – both on a local area network and in

long-haul nets. PVM [25] is often considered a de-facto standard in this field, and MPI [1] is emerging to replace PVM as a widely accepted standard. However, running very large problems with high resource requirements for both computation and communication is only about to be understood. Furthermore, most systems for network computing to date represent extremely complex implementations, and full understanding of the overheads, capacities, and granularities attendant in such systems is lacking.

In this report the disposal of the communication bottleneck of network environments is tackled. For communication performance in LAN-based networks, contention appears to become a limiting factor, whereas latency usually limits communication for WAN based networks. To ameliorate this situation, this work is focussed on a latency hiding technique [15] to be utilized by algorithms that allow for overlapping computation and communication. *Communication latency hiding* is defined informally as a technique to increase processor utilization by transferring data via the network and continuing with the computation at the same time. To analyze the opportunities and limitations of this technique, in this paper:

1. A model is proposed for the analysis of communication latency hiding by overlapping computation and communication in high-latency networks, and conclusions are drawn about the possibility to achieve a maximal gain.
2. A refinement of the TCP/IP protocol is introduced that implements efficient and deadlock-free message passing as well as communication latency hiding in UNIX environments. The introduction of a separate communication kernel serves as a basis for a novel architecture of software platforms for network computing.
3. Experimental results are presented that show that the model matches reality well enough to serve as a basis for qualitative prediction of performance gain due to latency hiding.

The field of scientific computing provides large classes of applications for which network computing can be effective. This paper begins with a typical technique found in many parallel implementations of scientific applications, domain decomposition [14], and a simple model of gain due to exploitation of communication latency hiding is presented. Then, a generic computational load measure is introduced that models a much broader class of algorithms. This theoretical analysis has motivated the development of a latency hiding protocol as a refinement of the TCP/IP suite [9]. By utilizing otherwise wasted protocol wait times for computational tasks, latencies are reduced further, and smaller granularities can be supported.

The earlier origin of this work was the development of PARC [22], a platform for network computing in the large. There, for the sake of portability, all protocols are implemented on top of TCP/IP and the Berkeley Socket Interface [20]. Stream sockets abstract from low level data transfer by offering the stream model to the programmer. However,

this does not suffice to exchange messages in a safe way, because deadlocks may occur due to bounded buffer implementations. The solution to this problem is left to the programmer. But not only with the socket interface, also IBM's external user interface [5] and Intel's NX message passing interface [18] burden the programmer with deadlock avoidance. Furthermore, message transfer might be inefficient due to protocol implementation details. The design decisions of the protocol, presented in this report, are partially based on empirical studies to ensure highest possible performance in a portable manner.

The model of communication latency hiding is based on ideas similar to those presented by Stone [21]. In the report to hand, the notion of software granularity and hardware granularity is introduced, which offers a simpler representation and broader range of conclusions. Further, this work generalizes the modeling of numerical applications, as was done by de Sturler and van der Vorst [24], and Wunderlich et al. [26], and describes the notion of gain in a broader context.

In Section 2, the model of communication latency hiding is introduced, and the implications for parallel processing are discussed. Section 3 presents an implementation of a latency hiding protocol for UNIX environments. Preliminary experimental results are used in Section 4 to validate the model and the protocol implementation.

2 A Latency Hiding Model

The idea of gain of communication latency hiding can be grasped by means of a simple model. First consider a concrete example from scientific computing; a two-dimensional parabolic partial differential equation that, for example, describes the phenomenon of heat conduction:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

Applying an explicit finite-difference approximation [19] yields the following formula, based on a five-point-stencil, also referred to as Jacobi-type iteration [6]:

$$u_{i,j}^{(k+1)} = r_x u_{i-1,j}^{(k)} + r_x u_{i+1,j}^{(k)} + (1 - 2r_x - 2r_y) u_{i,j}^{(k)} + r_y u_{i,j-1}^{(k)} + r_y u_{i,j+1}^{(k)},$$

where $r_x = \Delta t/(\Delta x)^2$, $r_y = \Delta t/(\Delta y)^2$, and $u_{i,j}^{(k)}$ denotes the temperature value at grid point $P_{i,j}$ and time $t_0 + k\Delta t$.

The index space of this equation is three-dimensional. Observing that $u_{i,j}^{(k+1)}$ depends only on values u of the previous time step k, storage requirements can be reduced by one dimension. Two two-dimensional arrays store the values of the entire spatial domain at two consecutive time steps $2k$ and $2k+1$. Newly calculated values of time step $2k+2$ are stored by overwriting the values of time step $2k$.

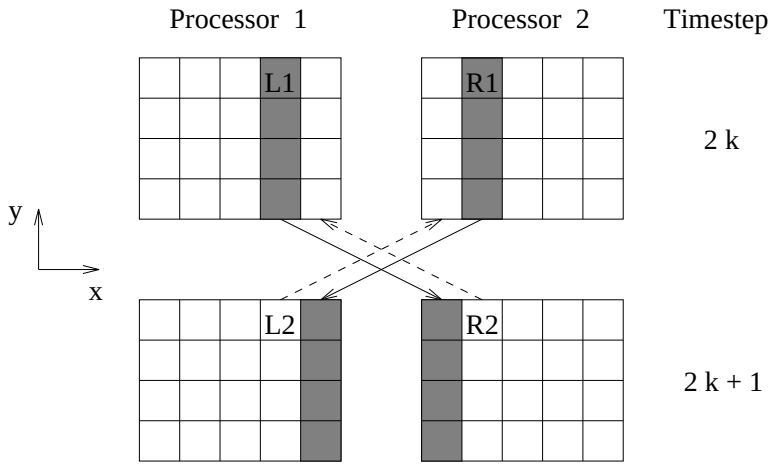


Figure 1. Domain decomposition.

For parallelization on a two-processor system, the spatial problem domain may be decomposed into two subdomains. These subdomains are overlapped according to the data dependencies of the five-point-stencil. Thereby, artificial boundary conditions are introduced that are exchanged between subsequent iterations according to Fig. 1. Starting the iteration with an initial condition, columns $L1$ and $R1$, calculated at time step $2k$, become the artificial boundary conditions of the other processor at the next time step $2k+1$. Thus, $L1$ and $L2$ have to be communicated before columns $L2$ and $R2$ can be calculated at time step $2k+1$. The implementation of this algorithm has been studied in detail on a LAN of workstations in [7].

2.1 A Simple Model

The purpose of the following model is a simple formalization of the phenomenon of communication latency hiding, and its effect on problem size and granularity. Assuming a runtime system that supports non-blocking send and blocking receive operations, the following abstract loop body of the time-iteration above would be straightforward to implement:

(i) Calculate all grid values
(ii) Send artificial boundaries to nearest neighbors $(A.1)$
(iii) Receive artificial boundaries from nearest neighbors

This loop body is separated into two distinct phases, the calculation phase comprises step (i), and the communication phase consists of phases (ii) and (iii). A simple model estimates the sequential execution time consumption t_{seq} of one loop iteration:

$$t_{seq} = C * n * m,$$

where n is the x-dimension and m the y-dimension of the spatial problem domain, and C is a machine dependent constant that denotes the average calculation time per grid point. Assuming an equal distribution of work among p processors, the calculation time per task is $t_{calc} = t_{seq}/p$, and the runtime of the parallelized loop body execution can be approximated by

$$t_{par}(p) = t_{calc} + t_{com} = \frac{t_{seq}}{p} + t_{com}.$$

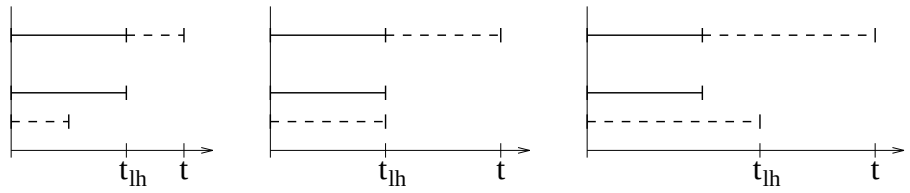
All overhead incurred is assumed to be communication related. The speedup for algorithm $(A.1)$ is

$$S(p) = \frac{t_{seq}}{t_{par}} = \frac{t_{seq}}{t_{seq}/p + t_{com}} = \frac{p}{1 + t_{com}/t_{calc}}$$

The ratio of calculation time and communication time per task is introduced as software granularity $\gamma = \frac{t_{calc}}{t_{com}}$. With γ, S can be written

$$S(p) = \frac{p}{1 + 1/\gamma}.$$

Comparing with Amdahl's law, $1/\gamma$ can be interpreted as the sequential portion of the program. To avoid its disastrous effect with respect to speedup, γ has to be large, i.e. acceptable speedup requires large software granularity. However, rather than restricting granularity γ to large values (and thus restricting ourselves to only large problem sizes) another possibility of reducing the (sequential) communication part is considered. The basic idea is *communication latency hiding* as illustrated in Fig. 2.



(a) $t_{calc} > t_{com}$ (b) $t_{calc} = t_{com}$ (c) $t_{calc} < t_{com}$

Figure 2. Illustration of communication latency hiding.

In this figure, the solid lines correspond to calculation time t_{calc}, and the broken lines indicate communication time t_{com}. In all three cases, the upper line shows the serialized execution of calculation followed by communication, as given by the parallelization scheme of algorithm $(A.1)$. The lower two lines illustrate calculation and communication starting at the same time, and executing concurrently, thus *hiding* the communication latency behind the calculation. The elapsed time t_{lh} of this version is always lower than the serial runtime t_{seq}. The three cases illustrate different degrees of latency hiding. In (a),

communication is hidden thoroughly, corresponding to $t_{lh} = t_{calc} > t_{com}$. Part (b) shows the equilibrium case, where calculation time equals communication time: $t_{lh} = t_{calc} = t_{com}$. In case (c), communication time exceeds calculation time, and thus cannot be hidden completely: $t_{calc} < t_{com} = t_{lh}$.

These considerations suggest that the performance of algorithm $(A.1)$ may be improved if communication and calculation are executed concurrently. The loop body of a modified algorithm $(A.2)$ is therefore:

(i) Calculate artificial boundaries of next time step
(ii) Send artificial boundaries to nearest neighbors $(A.2)$
(iii) Calculate grid values (except those done in step (i))
(iv) Receive artificial boundaries from nearest neighbors

To model the runtime of algorithm $(A.2)$, the overall domain is assumed to be sufficiently large, such that step (i) of $(A.2)$ can be neglected. Then, the idealized situation of Fig. 2 holds, and the parallel runtime is

$$t_{par}(p) = \max\bigl(t_{calc}, t_{com}\bigr).$$

The speedup of $(A.2)$ is derived analogously to $(A.1)$:

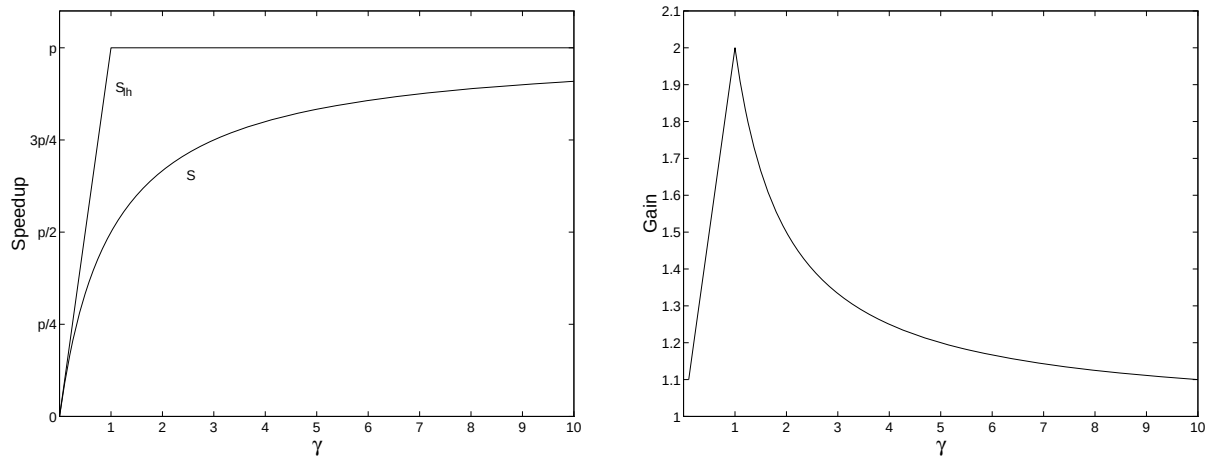
$$S_{lh}(p) = \frac{p}{\max(1, 1/\gamma)},$$

with γ and the same assumptions as for algorithm $(A.1)$. The ratio of the speedups of algorithms $(A.2)$ and $(A.1)$ is defined as the *gain*

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

G can be interpreted as speedup due to latency hiding, which only indirectly depends on p through γ. Gain G is furthermore directly related to efficiency. With the definition of efficiency $E(p) = S(p)/p$, the gain is the quotient of efficiencies $G = E_{lh}/E$. For parallel algorithms structured like $(A.1)$ and refined like $(A.2)$, gain G is bounded by the constant 2 independent of the number of processors being employed. However, it should be pointed out that without a high value of gain (i.e. close to 2), utilizing larger numbers of processors will in general lead to low efficiency. Therefore, gain itself is a secondary objective which will lead to higher efficiency, and leads to a solution that allows the use of a particular number of processors with maximum efficiency. It is this point that is built upon as a main thesis of this work. If large numbers of workstations are to be used, we are forced into high-latency environments for technical and economical reasons. Thus, techniques should

be used to exploit overlapping. Gain, as defined here, becomes a primary focus of the programmer and system designer, and therefore the relationships among gain, speedup, granularity (γ), and communication overhead are examined. Fig. 3(a) shows the speedup curves S and S_{lh} as a function of γ. Fig. 3(b) shows the corresponding gain G.



(a) (b)

Figure 3. Ideal communication latency hiding.

For $0 < \gamma \leq 1$, $G = 1 + \gamma$ is a linearly increasing function. This corresponds to case (c) in Fig. 2, where communication outweighs calculation. Note that for a smaller γ, the communication delay increases, thus less of it can be hidden, and the gain decreases. The maximal gain $G = 2$ is obtained with $\gamma = 1$, where the maximal amount of communication latency hiding occurs. For $\gamma \geq 1$, $S_{lh}(p) = p$ is constant. As $S(p)$ tends to p with increasing γ, $G = 1+1/\gamma$ decreases asymptotically towards the limit 1. The communication overhead, and thus the gain of latency hiding becomes negligible with increasing γ, i.e. communication time t_{com} is negligible compared to calculation time t_{calc} (cf. Fig. 2(a)).

The model of ideal communication latency hiding that lead to equation (1) can be easily generalized to algorithmic structures, where only a fraction of the calculation can be utilized for hiding communication. In algorithm ($A.2$), this part would be step (iii). Usually, the data that are exchanged between the cooporating tasks have to be calculated before the transfer. In the example of algorithm ($A.2$), this is done in step (i). Introducing the fraction f of the calculation time that is effectively available for hiding communication, the runtime of the parallel loop can be written as:

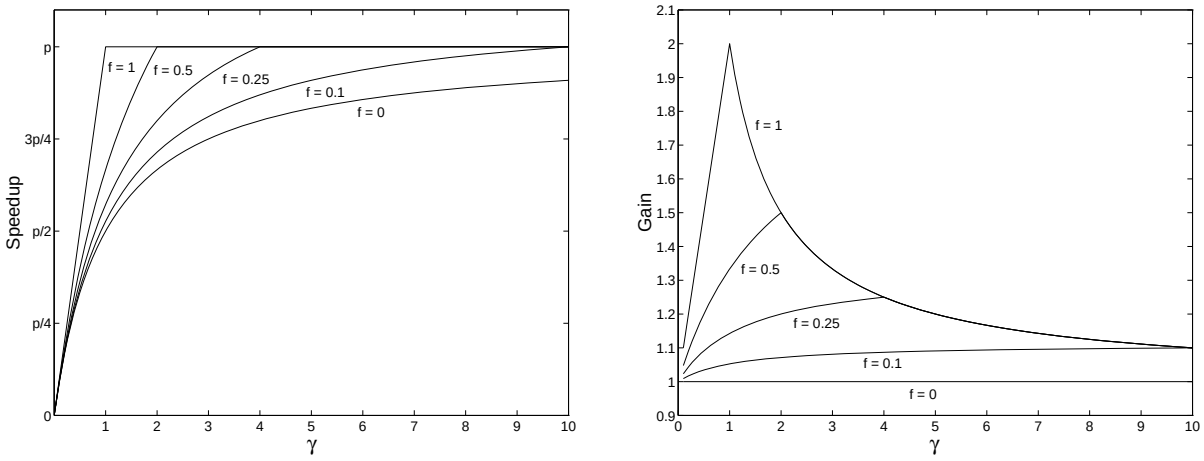
$$t_{par}(p) = (1 - f)t_{calc} + \max\big(ft_{calc}, t_{com}\big),$$

with $0 \leq f \leq 1$. De Sturler [23] uses this representation to analyze the speedup behavior

of Krylov subspace methods. The gain formula (1) extends to:

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

Obviously, $f = 1$ yields the ideal case modelled by equation (1), whereas for $f = 0$, no latency hiding is possible at all. In the following, communication latency hiding is called *ideal* if $f = 1$. Figure 4 illustrates the effect of fraction f on speedup and gain.



(a) (b)

Figure 4. Partial communication latency hiding.

This figure shows that even if the fraction of the calculation phase that contributes to communication latency hiding is relatively low ($f = 0.5$), the speedup reaches the maximal value p for a relatively low granularity γ. However, the maximal gain is only $G_{max} = 1 + f$ for $\gamma = 1/f$.

2.2 Extended Model

This section introduces an extension and refinement of the simple model. The goal is the incorporation of fundamental properties of the gain function as parameters. The starting point is the speedup formulae developed in the previous section:

$$(2) \qquad S(p) = \frac{p}{1 + t_{com}/t_{calc}}, \qquad S_{lh}(p) = \frac{p}{(1 - f) + \max(f, t_{com}/t_{calc})}.$$

These equations are refined by introducing (1) a more detailed description of the calculation and communication times, and (2) introducing a term that lumps together all overhead induced by the latency hiding implementation. The following assumptions are made: (1)

parallelization overhead other than latency hiding overhead can be gathered in t_{com}, and (2) all other effects not explicitly mentioned are negligible.

The *total calculation time* is modeled as

$$t = \frac{i}{r_{op}},$$

where i denotes the number of instructions needed to sequentially solve the entire original problem, and r_{op} the processor specific processing rate in instructions per unit time. Depending on the processor technology, r_{op} varies with respect to the program. However, r_{op} is assumed to be constant in the following.

The *message transfer time* can be described as follows:

$$t_{tr} = t_{cs} + \frac{b}{r_{com}} = \frac{i_{cs}}{r_{op}} + \frac{b}{r_{com}}, \tag{3}$$

where $t_{cs} = i_{cs}/r_{op}$ is the software overhead of the communication startup time, b the number of communication units per message, and r_{com} the network transfer rate with respect to this unit of communication. For small messages startup time is dominant, and for long messages, bandwidth (throughput) limits the transfer time. Although the network transfer rate is generally technology dependent, it often suffices to approximate r_{com} as a constant. From our experience, this assumption holds reasonably well in FDDI and Ethernet LANs, and even in heterogeneous WANs such as the Internet. Particular effects of different network technologies, such as congestion, can be modeled in r_{com}, for example by defining r_{com} as a function of the network transmission rate.

The *communication time* combines transfer time of a single message and the number of messages transferred subsequently during a communication phase:

$$t_{com} = q\, t_{tr}.$$

The number of messages q depends on the algorithm, and often on the number of processors p. Also, q depends on the hardware. It denotes the number of message transfers that are executed sequentially. In case, the hardware supports multiple network interfaces, and messages are transferred simultaneously via these interfaces, this would be reflected in the value q by counting simultaneous message transfers as $q = 1$. For example, a 2-dimensional torus topology with 4 independent hardware links, and a task that delivers four messages to its direct neighbors can be assumed to yield $q = 1$. However, if the same application were to run on an architecture with bus interconnect technology, all messages would have to be sent sequentially, yielding $q = 4$.

Granularity is the parameter used to characterize both a parallel application and a parallel machine. *Software granularity* γ_s and *hardware granularity* γ_h are distinctly defined as

$$\gamma_{s,j} \stackrel{\text{def}}{=} \frac{i_j(p)}{b_j(p)}, \qquad \gamma_{h,j} \stackrel{\text{def}}{=} \frac{r_{op,j}}{r_{com,j}}, \qquad 0 \le j < p.$$

The software granularity $\gamma_{s,j}$ of task j is the ratio of the number of instructions $i_j(p)$ of task j and the number of communication units $b_j(p)$, sent and received by this task. In this definition, $i_j(p)$ and $b_j(p)$ denote some function, which depends primarily on the number of processors p. If the partitioning of an algorithm matches a heterogeneous configuration, $\gamma_s = (\gamma_{s,0}, \ldots, \gamma_{s,p-1})^T$ might be modeled as a vector quantity, characterizing each task separately. Here, the simple case is studied, where the application is ideally parallelizable into p parts that are assumed to be identical in their calculation times and communication requirements. Assuming constant communication volume, the software granularity characterizes the parallel program with $i(p) = i_j(p) = i/p$ and $b(p) = b_j(p) = b$ that are identical for all j:

$$\gamma_s = \frac{i/p}{b},$$

where i denotes the number of instructions of the sequentialized parallel program. The hardware granularity $\gamma_{h,j}$ is the ratio of the instruction rate $r_{op,j}$ of processor j executing task j and the transfer rate $r_{com,j}$ of the network(s) processor j is connected to. In the following, the (collective) machine is assumed to be homogeneous, and the rates to be constant. Therefore, the hardware is characterized by the granularity:

$$\gamma_h = \frac{r_{op}}{r_{com}}.$$

Still neglecting the overhead of communication latency hiding, the speedup formulae (2) become

$$S(p) = \frac{p}{1 + q\left(\frac{i_{cs}}{i/p} + \frac{r_{op}}{r_{com}}\frac{b}{i/p}\right)} = \frac{p}{1 + q\left(\frac{i_{cs}}{i/p} + \gamma_h/\gamma_s\right)},$$

and

$$S_{lh}(p) = \frac{p}{(1-f) + \max\left(f, q\left(\frac{i_{cs}}{i/p} + \frac{r_{op}}{r_{com}}\frac{b}{i/p}\right)\right)} = \frac{p}{(1-f) + \max\left(f, q\left(\frac{i_{cs}}{i/p} + \gamma_h/\gamma_s\right)\right)}.$$

The gain of communication latency hiding (1) therefore becomes

$$G(p) = \frac{S_{lh}(p)}{S(p)} = \frac{1 + q\left(\frac{i_{cs}}{i/p} + \gamma_h/\gamma_s\right)}{(1-f) + \max\left(f, q\left(\frac{i_{cs}}{i/p} + \gamma_h/\gamma_s\right)\right)}.$$

A particular latency hiding approach will introduce its own overhead, which depends on the hardware and the implementation of the message passing protocol. In principle, overhead increases both the calculation and communication parts of a program. However, to include the effect of overhead, a term $\Omega \geq 0$ is added to the calculation part of the latency hiding version:

$$G(p) = \frac{1 + q\left(\frac{i_{cs}}{i/p} + \gamma_h/\gamma_s\right)}{(1-f) + \max\left(f + \Omega, q\left(\frac{i_{cs}}{i/p} + \gamma_h/\gamma_s\right)\right)}. \tag{4}$$

The term Ω models the qualitative overhead of communication latency hiding. Since accurate measures will be hard to obtain for quantitative modeling, we introduce a unitless granularity here to examine the effect of varying this parameter.

2.3 Discussion of Latency Hiding Gain

The discussion of the gain function G can be simplified by treating short messages and long messages separately. Short messages are characterized by $t_{cs} \gg b/r_{com}$, and long messages by $t_{cs} \ll b/r_{com}$ in equation (3). The gain then reduces to

$$G \approx \begin{cases} \frac{1+q\frac{i_{cs}}{i/p}}{(1-f)+\max(f+\Omega, q\frac{i_{cs}}{i/p})}, & t_{cs} \gg b/r_{com} \\ \frac{1+q\gamma_h/\gamma_s}{(1-f)+\max(f+\Omega, q\gamma_h/\gamma_s)}, & t_{cs} \ll b/r_{com} \end{cases} \tag{5}$$

For short messages, the startup times are predominant, for long messages, the size of the message accounts for the communication time that shall be hidden. Obviously, G shows the same behavior in both cases, when substituting $\frac{i_{cs}}{i/p}$ for ratio γ_h/γ_s. Since we are interested in huge problems with large communication requirements, we direct our attention to the second case with large messages. Here, hardware granularity, software granularity and latency hiding overhead each appear as parameters. To study their impact on the latency hiding gain, first the special case of a two-processor system is examined, and later extended to a more general multi-processor case.

2.3.1 Two-Processor System

To grasp the qualitative expressiveness of the latency hiding model, the following special case is considered:

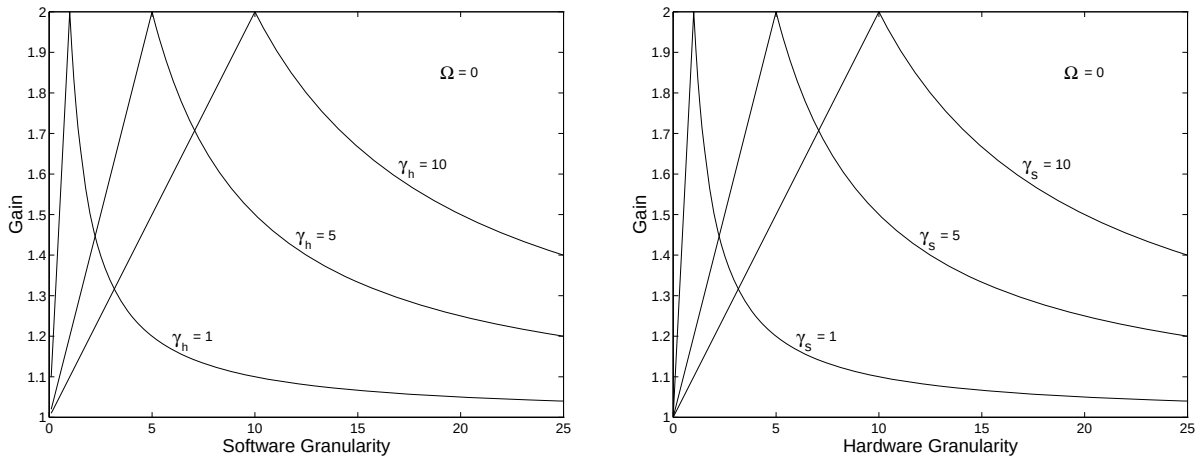
$$t_{cs} \ll b/r_{com}, \quad f = 1, \quad q = 1 \quad \Rightarrow \quad G \approx \frac{1+\gamma_h/\gamma_s}{\max(1+\Omega, \gamma_h/\gamma_s)}.$$

This means that within certain pairs of tasks, resulting from a pipe topology for example, one task sends a long message to the other task, and communication can be hidden ideally. Although G could be discussed in terms of the quotient γ_h/γ_s and the overhead only, the effects of hardware and software granularity would not become obvious. Varying γ_h and γ_s corresponds to different interpretations. Changing γ_s can be interpreted as working with a fixed amount of computation, and changing the communication volume, or as assuming a fixed communication volume and changing the amount of computation. Varying γ_h corresponds to changing processing elements, network technologies or implementations of software communication interfaces. In the following, both parameters are treated explicitly.

Figure 5 shows a family of gain curves versus software granularity and hardware granularity for negligible overhead $\Omega = 0$. The curves have identical shapes. Due to the quotient γ_h/γ_s, points lying on the linearly increasing part of a curve in one figure correspond to points on the decreasing part in the other figure. G reaches its absolute maximal value of 2 for $\gamma_h = \gamma_s$. Two conclusions can be drawn from these figures:

1. Larger granularities provide a certain degree of robustness concerning the range that provides a particular gain, both in a hardware and software sense. For example, consider a desired lower gain bound of 1.5. Given a hardware granularity of $\gamma_h = 1$, the software granularity must be within the range $0.5 \leq \gamma_s \leq 2$. However, for $\gamma_h = 10$, the range increases to $5 \leq \gamma_s \leq 20$.

2. Smaller software granularity shifts the point of maximal gain towards smaller hardware granularity, and analogous, smaller hardware granularity shifts the point of maximal gain towards smaller software granularity. This result coincides with the need of faster communication networks for programs with smaller grain size. However, Fig. 5(b) also shows that smaller hardware granularities may not be necessary for applications with a lower bound of software granularity.



(a) (b)

Figure 5. Gain without overhead ($\Omega = 0$).

Figure 6 illustrates the effect of overhead on the gain G. Four insights can be extracted from this figure:

1. Increasing overhead results in a decreasing maximal gain (≤ 2).

2. Increasing overhead results in a smaller range of software and hardware granularities, within which a particular gain can be obtained.

3. To gain from latency hiding, the condition

$$\gamma_h/\gamma_s > \Omega$$

must hold. The latency hiding overhead Ω affects G, if $\gamma_h/\gamma_s < 1 + \Omega$, according to the maximum function in the denominator of G. Thus, the gain becomes a loss, if $G = \frac{1+\gamma_h/\gamma_s}{1+\Omega} < 1$, i.e. if $\gamma_h/\gamma_s < \Omega$. This condition puts constraints on the application granularity, as well as the hardware designer, system software and language designer to keep γ_h/Ω large. Because it is intuitive that γ_h should not be increased, Ω must be therefore minimized.

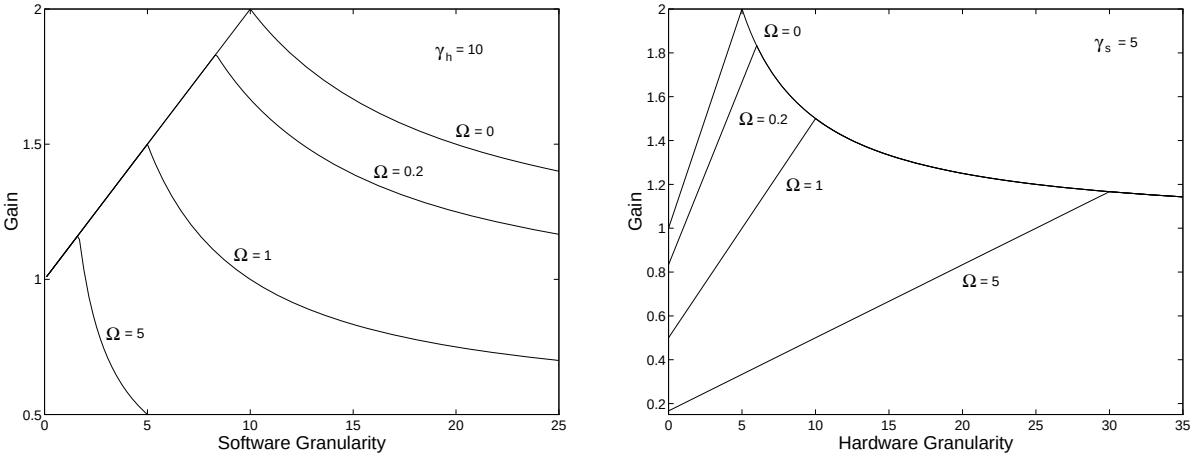
4. The fact of most importance to the parallel programmer is the point where gain G is maximal:

$$1 + \Omega = \gamma_h/\gamma_s.$$

Under this condition, the maximal gain becomes

$$1 \quad < \quad \max(G) = 2 - \frac{\Omega}{1+\Omega} \quad \leq \quad 2.$$

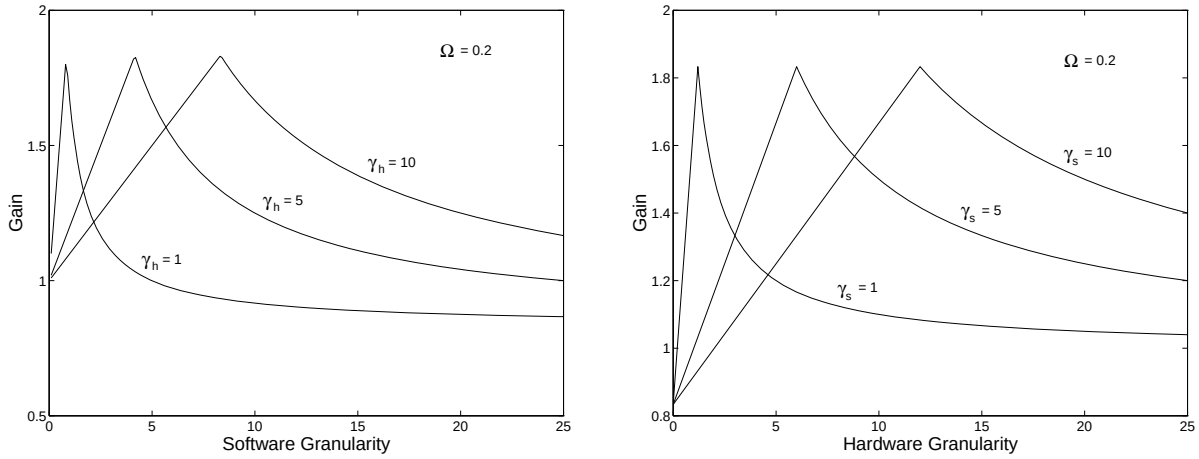
This dependency of the maximal gain on overhead, as well as hardware and software granularity, can also be readily observed in Fig. 6.



(a) (b)

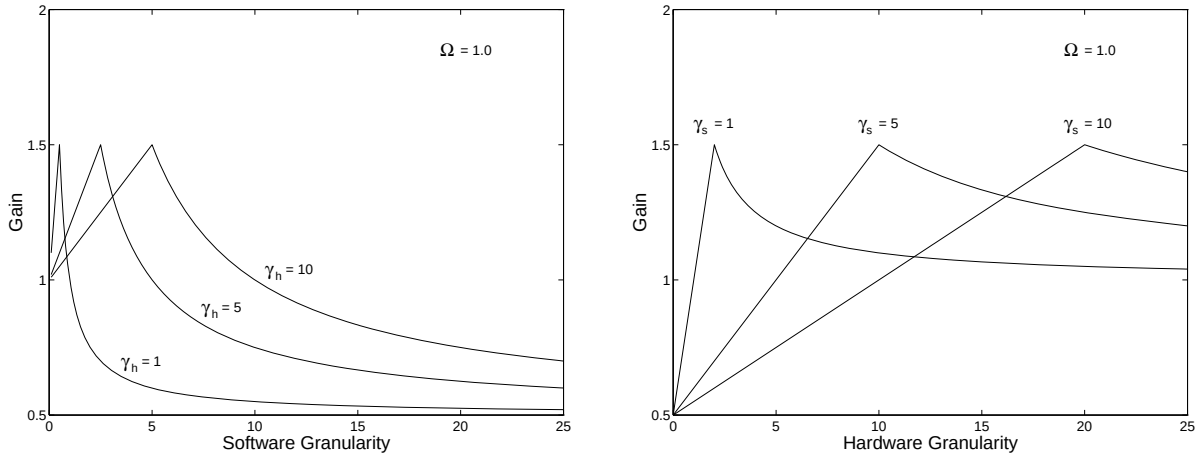
Figure 6. Gain dependency on overhead.

Figures 7 and 8 show the impact of varying overhead on gain G. Two examples shall illustrate the use of these figures.



(a) (b)

Figure 7. Gain with overhead ($\Omega = 0.2$).



(a) (b)

Figure 8. Gain with overhead ($\Omega = 1.0$).

Example 1 Assume a given software granularity $\gamma_s = 5$, and that three different architectures are available, characterized by the following pairs (γ_h, Ω): $H_1 = (1, 1), H_2 = (5, 0.2), H_3 = (10, 0)$. In Figures 8(a), 7(a), and 5(a), the respective gains can be found: $G(H_1) = 3/5$, $G(H_2) = 5/3$, $G(H_3) = 3/2$. It follows that the given application achieves the best gain on architecture H_2.

Example 2 Given a range of software granularities $2 \leq \gamma_s \leq 10$. Which of the architectures $H_1 = (5, 0), H_2 = (1, 0.2), H_3 = (10, 1)$ guarantees an effective gain $G > 1$ for the entire range? Figures 5(a), 7(a), and 8(a) show that architectures H_1 and H_3 fulfill the requirement, while H_2 does not.

2.3.2 Multi-Processor System

From approximation (5), we see that for large messages gain G depends only indirectly on the number of processors p through the software granularity $\gamma_s = \frac{i(p)}{b(p)}$. The discussion of the two-processor system was based on the assumption that $i(p) = i/p$ and $b(p) = b$, where the number of instructions i and the size of the messages b is constant. In this case, the dependency of software granularity upon the number of processors can be made explicit by introducing the ratio

$$\lambda = p\frac{\gamma_s}{\gamma_h} = \frac{i/b}{\gamma_h}.$$

Then, gain function (4) becomes

$$G(p) = \frac{1 + pq\left(\frac{i_{cs}}{i} + 1/\lambda\right)}{(1-f) + \max\Big(f + \Omega, pq\left(\frac{i_{cs}}{i} + 1/\lambda\right)\Big)}.$$

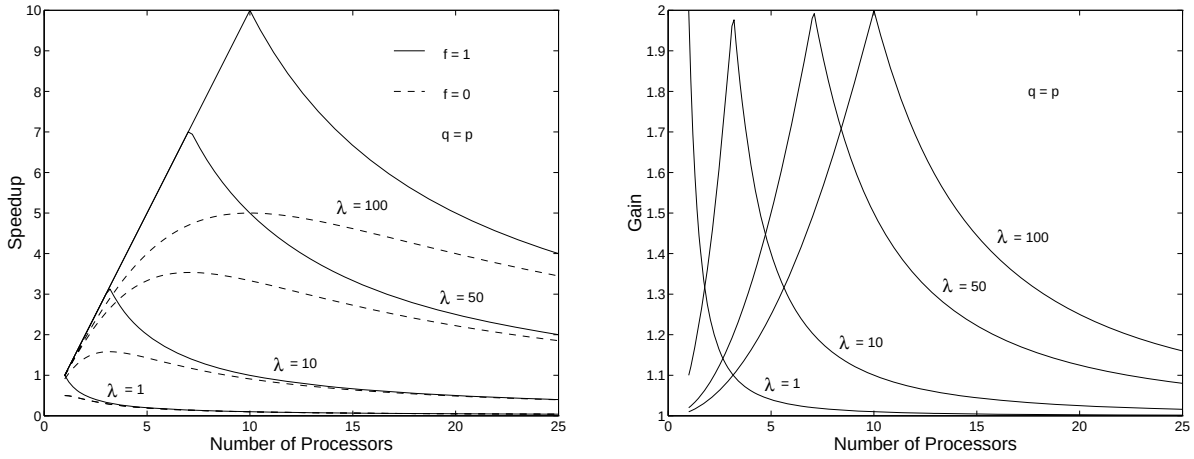
Focusing on long messages, startup times are neglected, and the formulae for speedup (2), including latency hiding overhead Ω, turn into

$$S(p) = \frac{p}{1 + qp/\lambda},$$
$$S_{lh}(p) = \frac{p}{(1-f) + \max\Big(f + \Omega, qp/\lambda\Big)},$$

and gain $G(p) = S_{lh}(p)/S(p)$:

$$G(p) = \frac{1 + pq/\lambda}{(1-f) + \max\Big(f + \Omega, pq/\lambda\Big)}. \tag{6}$$

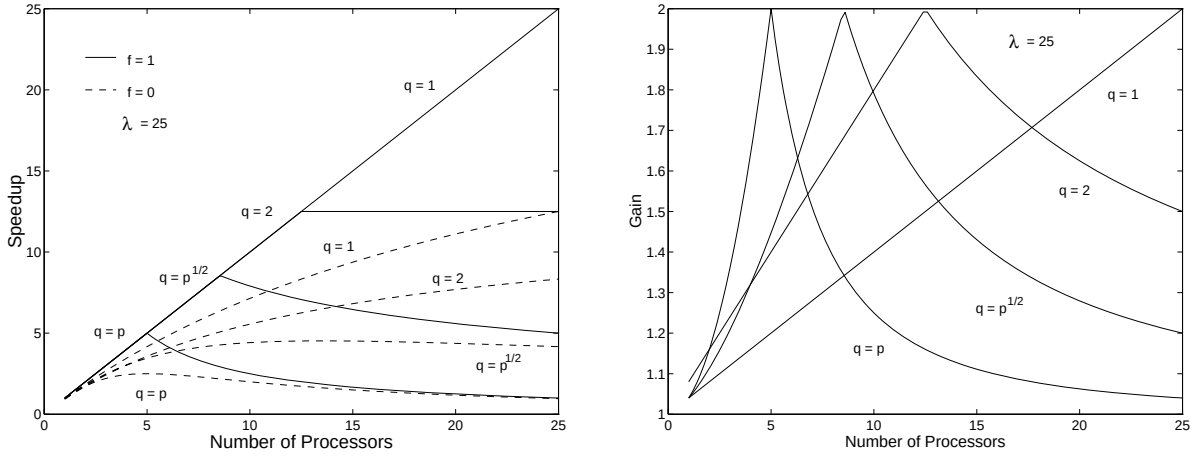
Figure 9.1 illustrates the dependency of gain on the ratio of software and hardware granularities λ. In contrast to the previous discussions where $q = 1$, the communication volume is now assumed to be proportional to the number of processors ($q = p$), which corresponds to all-to-all communication. The gain values in 9.1(b) correspond to the case of ideal latency hiding ($f = 1$) with respect to the case without latency hiding ($f = 0$). Figure 9.1(a) shows the speedup curves $S_{lh}(p)$ of both versions. The dashed curves belong to the case without latency hiding. Because the communication volume is proportional to the number of processors, the dashed speedup curves attain a global maximum. Adding processors beyond the number corresponding to p_{opt}, leads to smaller speedup values, because communication overhead outweighs calculation time.



(a) (b)

Figure 9.1. Gain dependency on granularity λ ($\Omega = 0$).

Figure 9.1 shows that the maximal value of speedup with ideal latency hiding is 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. For sensible utilization of large numbers of processors, λ has to be big. In case of $q = p$, and the idealized situation $f = 1$, $\Omega = 0$, the maximal gain $G = 2$ is obtained from equation (6) for $\lambda = p^2$.



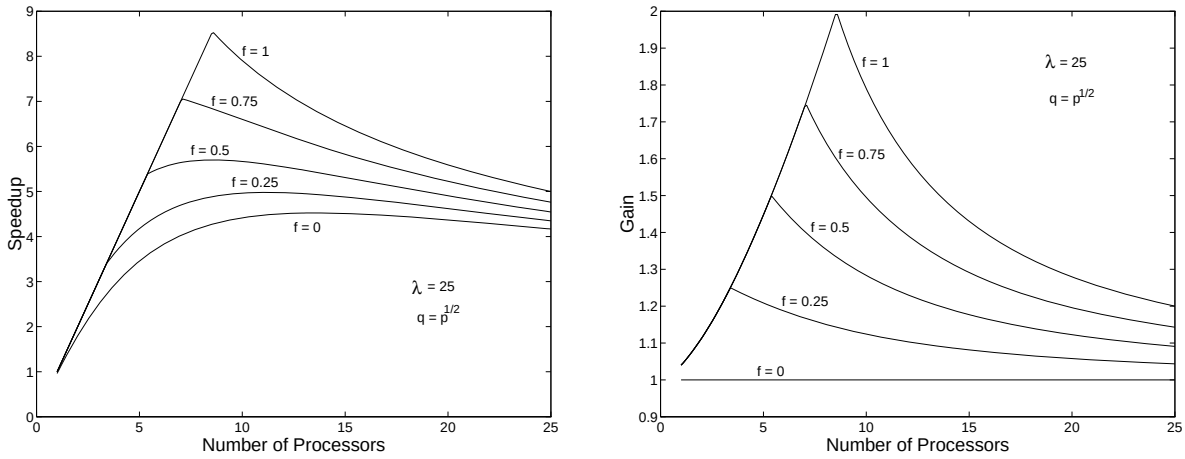
(a) (b)

Figure 9.2. Gain dependency on communication volume q ($\Omega = 0$).

Figure 9.2 shows the effect of varying the number of messages q on speedup and gain. The ratio of software and hardware granularities is fixed. It is interesting to note that the

speedup of the version with ideal latency hiding does not decrease with increasing number of processors, if the communication volume is constant (cf. $q = 2$ in Fig. 9.2). This case is equivalent to the simple model presented in Fig. 3. Furthermore, as expected, increased communication volume leads to smaller maximal speedup values, and smaller numbers of processors where this maximum is achieved.

Analogous to the two-processor system, the influence of overhead can lead to a loss ($G < 1$) instead of a gain if $\Omega > pq/\lambda$. From equation (6) follows that this result is independent of the degree of latency hiding f. A reasonable implementation of communication latency hiding must induce an overhead $\Omega \ll 1$. This is possible in practice as will be shown in the next sections. So, there will be an effective gain, which implies that communication latency hiding can improve both speedup and efficiency.



(a) (b)

Figure 9.3. Gain dependency on latency hiding degree f ($\Omega = 0$).

Figure 9.3 shows the impact of varying the fraction of calculation that is available for hiding communication on speedup and gain for fixed overhead, granularities and number of messages. As before, the speedup values for $f = 0$ correspond to the algorithm without latency hiding, and are therefore the reference values for the gain calculation. For $f = 1$, ideal latency hiding is given.

Figure 9.3(a) depicts a typical speedup curve of an application without latency hiding ($f = 0$), where the communication volume increases proportional to the number of processors, in this example $q = \sqrt{p}$. Figure 9.3(a) illustrates, how the maximum of speedup increases and shifts towards smaller numbers of processors with increasing degree of communication latency hiding. These curves clearly demonstrate the benefit of communication latency hiding: Maximal speedup and efficiency are increased simultaneously with increasing f. Note that the points of maximal speedup and maximal gain do not coincide.

2.3.3 Impacts on Parallel Processing

The modeling rises several basic questions concerning communication latency hiding that can be of interest for different communities:

Architect: What is the best hardware granularity for a fixed overhead Ω?

There is a trade-off between the cost for smaller hardware granularity and the lower bound of software granularity that delivers maximal gain.

Programmer: (1) Assuming a given machine (γ_h, Ω), what is the optimal software granularity to achieve maximal speedup through latency hiding? (2) Given a certain application with software granularity γ_s, what is the best available parallel machine (γ_h, Ω) to optimize the speedup through latency hiding?

The programmer can use the model to determine the best software granularity, if this choice is open with respect to the application requirements. More important, the programmer can choose that machine among all accessible ones, that yields the highest speedup. Due to the simplicity of the expression $G = S_{lh}/S = t_{par}(p)/t_{lh}(p)$, where $t_{par}(p)$ and $t_{lh}(p)$ are the runtimes of algorithms $(A.1)$ and $(A.2)$ with p processors respectively, just two experiments are necessary to determine the actual gain G.

System software designer and language designer: How can the overhead Ω be reduced best?

This question depends on many parameters. In the following section, a concrete implementation for UNIX environments is introduced, illustrating the parameters arising in this setting. In Section 4, results are discussed with implementations in a UNIX environment and a Transputer based multi-processor.

3 Latency Hiding in UNIX

Communication latency hiding can be provided by building additional communication processors into the hardware. This technology is the basis of the more recent generations of parallel architectures such as the Intel Paragon. Communication latency hiding can also be emulated on workstations by refining the send and receive operations implemented on top of the TCP/IP protocol. The ideas about latency hiding arose during the design of a message passing kernel, which is the basis of PARC [22], a platform for network computing in the large. To ensure portability, all protocols are implemented on top of TCP/IP and the Berkeley socket interface [20]. Although stream sockets abstract from low level data transfer by offering a stream model to the programmer, there are still deficiencies with messages passing. Deadlocks may occur due to bounded buffer implementations, or the transfer becomes unnecessarily inefficient due to implementation details. This section describes how latency hiding, combined with efficient and deadlock-free communication can be realized in UNIX environments.

3.1 Aspects of UNIX Interprocess Communication

UNIX provides the system calls `read` and `write` (or, almost identical, `send` and `recv`) to communicate across connection-oriented communication channels by means of streams [20]. A contiguous memory region is sent as a stream of bytes, and received as a stream of bytes. Composing and decomposing the stream is in the programmer's responsibility, for both inserting and extracting individual data items from the stream and for determining message boundaries. If message lengths are known at the sender and receiver side, the stream can correctly be split into messages, because the connection-oriented protocol ensures the stream to be unique. If the length of a message is not known a priori at the receiver end, the sender can prepend it in a message header.

For the following discussion the overloading of terms used for the description of message passing shall be resolved. In the classical theory of communicating processes, blocking denotes that the send or receive operations wait until the operation has been terminated. A non-blocking send operation returns as soon as the send call has been submitted to the underlying system layer independent of whether the data have been transferred, and a non-blocking receive call returns whether data are available or not. Synchronous communication refers to the case, where the send *and* receive operations block, until the data are transferred. Asynchronous communication denotes the three cases where at least one of the communication calls is non-blocking. Within this classification, UNIX communication is asynchronous on the application level. In contrast to the former definition, the terms blocking and non-blocking denote the operational behavior of the communication primitives on the transport layer. The send call can return after having transmitted only a part of the message, and the receive call after having received only a part of the message. This is in contrast to the classical semantics, where the transmission is usually viewed as atomic, and either the whole message is sent or received or nothing.

3.1.1 Message Buffering

For message passing on top of streams, the message data have to be provided in a contiguous memory region. Therefore, structured data or different values scattered over the memory have to be marshaled into a continuous buffer[1]. On the receiving side, a buffer is needed to receive the message. An ideal buffer management would allocate buffer memory and marshal the data in negligible time. In practice, this is not the case.

---

[1] UNIX offers scatter read and gather write system calls `readv` and `writev`. Although these operations are supported by network hardware, they are not portable [20], and assigning buffer start addresses and message lengths also incurs overhead.

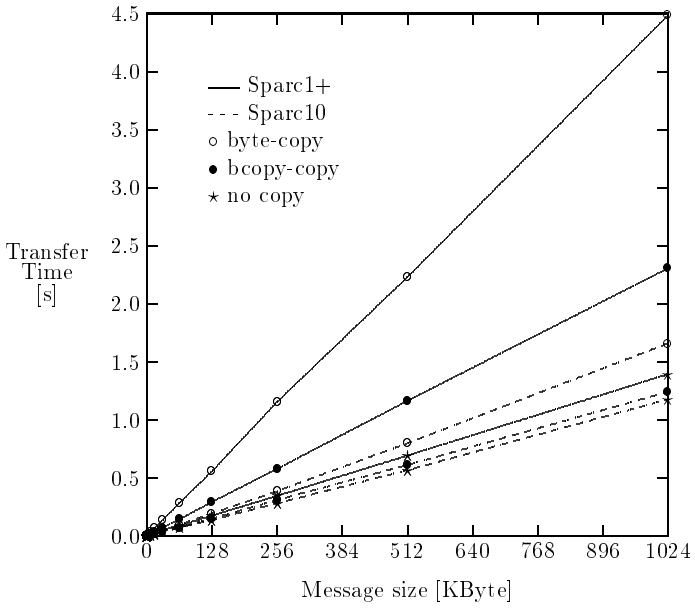


Figure 10. Overhead of message buffering.

Figure 10 shows message transfer times versus message size measured between two Sun SPARCstation1+ and two SPARCstation10 connected by an Ethernet. Three modes of buffering are shown: (1) byte-wise copying with an explicitly programmed for-loop, which is representative for the cost of data representation conversion; (2) `bcopy` is a C-library function for byte-wise memory-to-memory copying; (3) without copying into or out of a message buffer. Relatively fast machines like a Sun SPARCstation10 perform the copying in negligible time compared to the transmission time of an Ethernet. In contrast, a Sun SPARCstation1+ needs more than three times the transmission time to copy data into and out of a message buffer with the explicitly programmed copy-loop. However, although the machines deliver different peak performance, both are fast enough to operate at nearly the same transmission rates without copying, identifying the Ethernet as the limiting component. In principle, the lower the hardware granularity, the more essential is an efficient buffer management for communication performance. Avoiding message copying at all has been proposed repeatedly [8,11,12].

In order to design a deadlock-free, efficient message passing protocol on top of UNIX stream sockets, the buffer management of asynchronous communication is investigated in more detail. The sockets at both ends of a connection-oriented channel allocate a receive buffer and a send buffer. In principle, two cases have to be distinguished: The sum of the lengths of the send buffer at the sender side and the receive buffer at the receiver side is (1) smaller than the message, and (2) not smaller than the message. In the first case, a blocking send call will not terminate, until the receiver empties the receive buffer. This situation can lead to deadlocks, if two processes simultaneously send messages to each

other. With the default buffer size of 4 Kbyte, messages bigger than 8 Kbyte cause both processes to deadlock in the send operation. Deadlocks cannot occur in the second case.

However, for performance considerations, it is interesting to investigate two subcases of case (2): (a) the message is smaller than the send buffer, (b) the message is larger than the send buffer. In case (a), the send call returns as soon as the entire message has been copied into the socket buffer. While the user process continues, the data are communicated by the TCP protocol. Case (b) is more complex. Suppose for simplicity that send buffer and receive buffer have the same size. Now, the first part of the message is copied into the send buffer, and communicated to the receive buffer. The send call returns after the second part of the message is copied into the send buffer. If the receiver has not yet submitted a receive call, its receive buffer is still fully occupied and the data in the send buffer at the sender side cannot be transferred. Within the TCP protocol layer, the receiver advertises a zero window, and the sender starts probing the receiver periodically [9]. Handling this case is expensive, and eventually increases the runtime of the send call dramatically. Only after the corresponding receive call has emptied the receive buffer, the pending send buffer content can be transferred. This again increases the cost of the receive call, because the message is not entirely available in the receive buffer yet, but is partially transferred during the receive call.

An implementation detail of the TCP protocol leads to increased transfer times, if the message size ranges from 4097 to 5552 bytes. Expected times of less than $10\,ms$ on an Ethernet jump up to about $100\,ms$. This can be explained historically, when small messages have been buffered until one bigger message could be sent, or a timer expires [16]. On Sun systems, this effect has been removed for small messages, but not for "4 KByte + small messages". For "8 KByte + small messages" the increased transfer times also appear, but only about 50% bigger than the expected transfer times. Switching on the TCP_NODELAY option removes this peculiarity. Similar observations have been reported by Crowcroft et al. [10].

3.1.2 Message Fragmentation

Streams relieve the programmer from implementation details such as fragmenting messages into network dependent maximal transmission units [20]. TCP/IP hides these details of interhost communication. Surprisingly, fragmenting messages above the TCP/IP transport layer can reduce communication latency as shown in Fig. 11. The measurements were conducted on two SPARCstation1+, connected via an Ethernet. Obviously, there exists a local minimum of transfer time for a fragment length of 4 Kbyte. This suggests that transmission of 4 Kbyte fragments is the best way of communicating long messages in UNIX environments. Although for large messages ($>$ 16 Kbyte in Fig. 11), transfer without fragmentation is slightly faster, it is not safe, since deadlocks may occur, if the message

size exceeds the maximal socket buffer size.

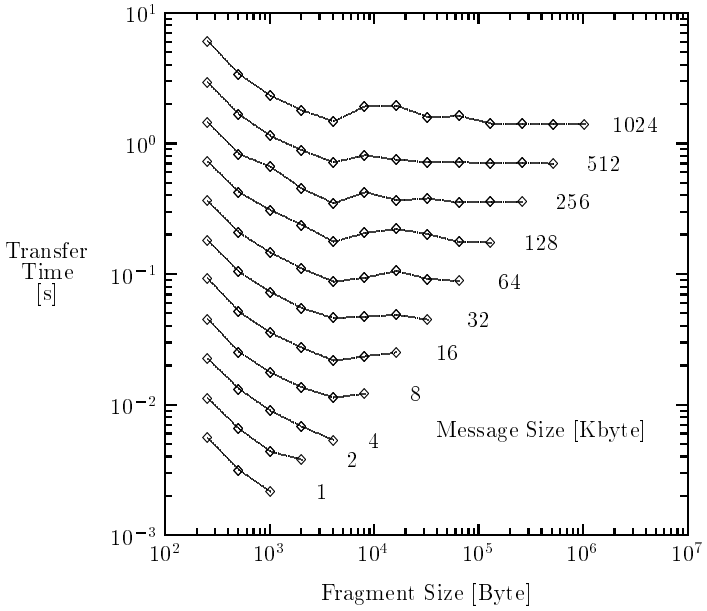


Figure 11. Performance of message fragmentation.

The reason for the optimum fragment size of 4 Kbyte is an implementation detail of the dynamic buffer memory management of UNIX [16]. An *mbuf* memory buffer stores 112 bytes of data. To buffer larger messages, multiple *mbuf* data structures are connected as a linked list. If the message length exceeds half of the machine's page size, a whole page is mapped into an *mbuf* structure. In contrast to managing a linked list of mbufs, the page-table entry is cheaper. Data are moved without memory-to-memory copies by remapping pages.

3.1.3 Deadlock Avoidance

Asynchronous communication requires buffering of data at the receiver side. Because these buffers are bounded in UNIX, deadlocks may occur, if the communication system calls are blocking. Suppose, two processes execute a send operation, sending a message to each other, before they invoke a receive operation to copy the received message out of the TCP receive buffer. As mentioned above, if both messages are larger than the capacities of the corresponding send and receive buffers, both processes block in their send calls — the processes deadlock. Pierce discusses this, and a more complicated example, for the Intel NX message passing system [18]. Bala et al. [5] describe how to write a *safe* program with the IBM external user interface. However, in both systems, deadlocks can occur, and the programmer is responsible for deadlock avoidance. For UNIX stream sockets, the default

send and receive buffer size is 4 Kbyte. Although this maximal buffer size can be adjusted with the `setsockopt` system call[2], bounded buffers are a principle problem.

A solution for avoiding deadlocks when communication via 4.3BSD sockets is sketched here informally. The key is fragmenting messages into sizes not bigger than the send or receive buffers, and reading from a socket with pending data in the receive buffer before writing any data. Multiple processes sending a message consisting of multiple fragments to each other, first read an eventually arrived fragment. This frees receive buffer space that the other process might use to complete the send operation. The total ordering of respective simultaneous send and receive operations guarantees the correctness of this algorithm. A formal treatment is beyond the scope of this report and appears elsewhere.

The correctness and efficiency criteria developed in this section imply the following design decisions for a message passing protocol:

1. To increase efficiency, message copying is avoided where possible.
2. To avoid deadlocks, messages are fragmented above the TCP transport layer into pieces not larger than the socket's send and receive buffers.
3. For efficiency reasons, the fragment size is fixed at 4 Kbyte. This size coincides with the fragment size required in 2. for default socket buffer sizes.

3.2 Protocol Support for Latency Hiding

The basic concept for implementing communication latency hiding in UNIX is the separation of communication and calculation into two processes. The communication process contains the message passing protocol, and the calculation process executes the user program with special communication primitives.

3.2.1 Overlapping Calculation and Communication

In this section, the ideas based on the observations in the previous sections are introduced stepwise. Consider the application described in Section 2 (Fig. 1). The implementation of the loop body given in algorithm ($A.1$) is shown in the following pseudo code fragment ($I.1$) running on two processors:

```
Calculate entire domain
write(s, abcol, len);                                   (I.1)
read(s, bcol, len);
```

In this code fragment, `abcol` is a pointer to the artificial boundary column, that is calculated as part of the domain and serves as boundary condition during the calculation of

[2] The maximal receive buffer size of 4.3BSD is 52,428 bytes.

the next time step. `bcol` points to the boundary column, that stores the received artificial boundary calculated in the current time step. The socket `s` denotes the communication end point of the UNIX stream connection, and `len` the number of bytes to be sent or received, respectively.

Figure 12 illustrates the runtime behavior from the user's point of view (a), and relevant system details in (b). The communication and calculation structure in this, and several following figures, is depicted by means of event diagrams. The send and receive buffers of the sockets are assumed to be of equal size, and message size is twice the buffer size. Therefore, the solid lines correspond to the transfer of a message fragment size equal to the buffer size. The broken lines symbolize acknowledgments. TCP internal fragmentation is omitted for clarity.

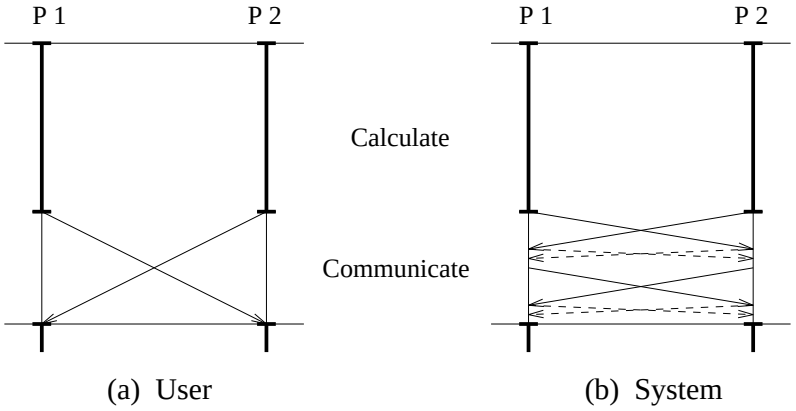


Figure 12. Naive implementation (A.1).

As mentioned earlier, this kind of asynchronous message passing deadlocks with blocking send primitives, if the message sizes are larger than the sum of the corresponding send and receive buffer lengths. The only safe way to avoid deadlocks is to write two codes, one for each process, with a reversed order of the send and receive calls. As shown later, the resulting ping-pong styled communication structure requires not only more programming effort, but is much less efficient than an interleaved, simultaneous communication.

The implementation of algorithm ($A.2$), requires only little restructuring of the code fragment ($I.1$):

Calculate artificial boundary
`write(s, abcol, len);` ($I.2$)
Calculate interior domain
`read(s, bcol, len);`

To maximize the overlap of communication and calculation, the artificial boundary is calculated first, and transmitted. During the main part of the calculation, the computation

of the inner points of the domain, the message can travel over the network, before it is received by the user process. This notion is illustrated in Fig. 13(a). Figure 13(b) shows the lower level details as described already in Section 3.1.1. The first message fragment fills the receive buffers. The second fragment cannot be transferred before the receive calls have emptied the receive buffers.

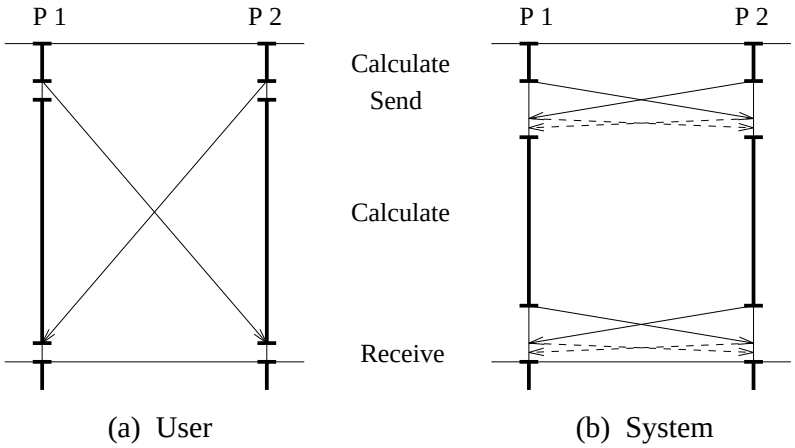


Figure 13. Separation of send and receive (A.2).

The problem of this implementation is the cost associated with the blocking send and receive calls. In fact, this version can even be less efficient than the naive implementation, if the blocking times introduce a significant overhead. Using non-blocking calls here is no alternative. Even worse, it would burden the programmer with keeping track of the number of bytes consumed by the transport layer.

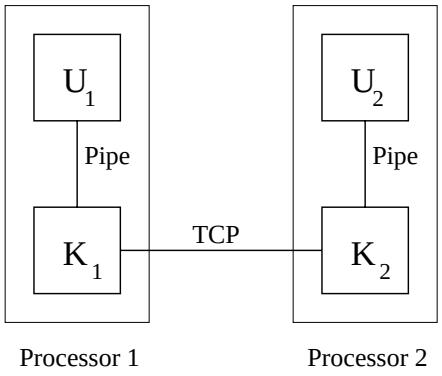


Figure 14. Process separation: User process and communication process.

A way to utilize the CPU idle times during blocking send operations is the introduction of a separate communication process. This setting is shown in Fig. 14. The

communication processes K_1 and K_2 are connected through a TCP channel. The user processes U_1 and U_2 communicate via these communication processes. The corresponding user and communication processes can exchange data via a pipe.

The effect of process separation on the interleaving of communication and calculation is shown in Fig. 15. Once a message to be sent is available to the communication process, it starts submission, and returns the CPU to the user process as soon as the transport layer idles. The calculation is interrupted by the communication process if a TCP event occurs. The overhead of context switching becomes less significant, with larger hardware granularity.

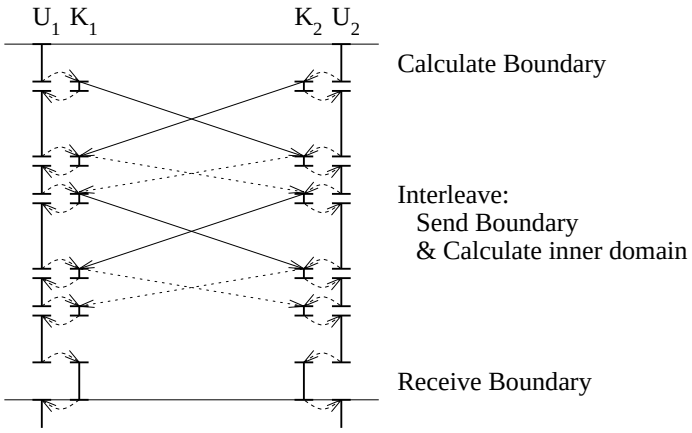


Figure 15. Context switching for communication latency hiding.

This mechanism is well suited for large messages. For very small messages with dominant message startup times, the time for context switches between user and kernel process should be much smaller than the time for message startup. To keep the overhead of the two-process implementation small, also an efficient transfer of the message between user and kernel process, as well as their efficient synchronization are necessary. A solution based on shared memory, together with the necessary program notation is presented in the next section.

3.2.2 Synchronization and Message Access

Given the process splitting described above, the main implementation objective is to minimize the latency hiding overhead Ω introduced in Section 2. For the following discussion, the implications of embedding latency hiding into the PARC system [22] are considered. This configuration comprises an extended communication process, called *kernel process* that is not only responsible for communication, but also for dynamic process generation and termination. The kernel process spawns a process that executes a user program.

Two mechanisms are necessary to implement latency hiding in this setting. Both processes, user process and kernel process, must be able to access a message, and both processes must be able to synchronize. These mechanisms are required to be fast to minimize overhead. To access the message, it could either be communicated between the processes, or be referenced in a physically shared memory segment, as for instance provided by UNIX System V IPC [20]. The latter mechanism is faster, because it avoids copying and communicating the message between user process and kernel process. Additional expense such as allocation of shared memory segments and portability problems are therefore taken into account. Fortunately, they can almost be hidden from the programmer.

More intricate is the choice of a synchronization mechanism for user and kernel process. The alternatives are either System V semaphores, or synchronizing messages. Synchronizing two processes by means of semaphores requires the execution of the semaphore operations, and switching the context of these processes.

Machine Model	Sem Sync [ms]	Pipe Msg [ms]
HP9000	0.09	0.18
IBM RS6000	0.053	0.050
SGI Crimson	0.222	0.222
Sun SPARCstation1+	0.5	0.38
Sun SPARCstation10	0.19	0.11

Table 1. Overhead of synchronization.

When a new user process is generated by PARC [22], it is automatically connected to the kernel process via a bidirectional pipe (two unidirectional UNIX pipes [20]). This connection suggests implementing synchronization by means of a pipe messages. Communication via a pipe should in principle require more time than a semaphore synchronization. Surprisingly, this is not always the case. Table 1 lists the cost of a single synchronization by means of semaphores, and transferring an 8 byte message (comprising a 4 byte header and 4 byte message identifier) via a pipe for various machines. This table shows that synchronization my means of semaphores is more expensive than transferring a short message via a pipe on Sun SPARCstations and IBM RS6000. Only the HP9000 workstation offers faster semaphore operations. Figure 16 shows the costs of semaphore synchronization and synchronization by means of pipe communication.

The plot symbols of Fig. 16 at message size 1280 Byte, and labelled with the machine types mark the measured synchronization times of two processes by means of semaphores. The lines marked with the same plot symbols show the transfer times of unbuffered messages via a pipe on the respective machines. This figure lead to the design decision that synchronization will be done by exchanging short messages via the pipe. For example, transferring about 512 Byte via a pipe on a SPARCstation1+ is almost as expensive than

the synchronizing overhead by means of a System V semaphore. To implement synchronization with semaphores, an appropriately large set of semaphores is needed, corresponding to the number of concurrently pending messages, and faster hardware support to reduce the synchronization overhead.

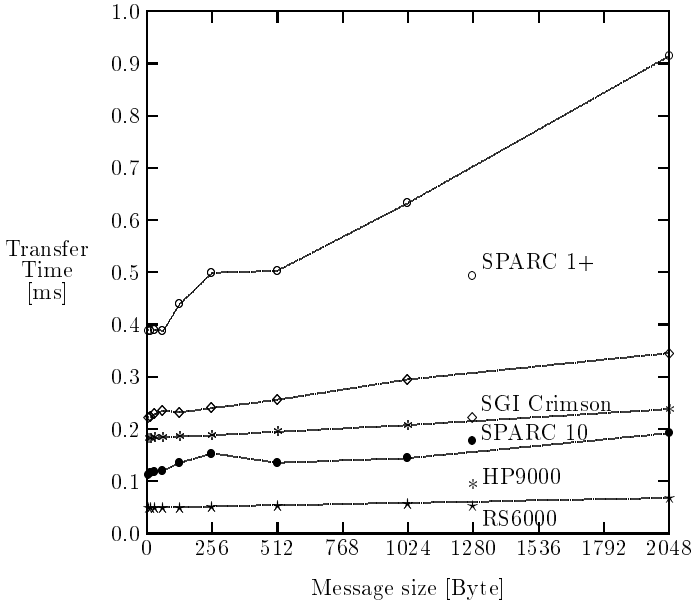


Figure 16. Semaphore synchronization versus pipe communication.

Figure 17 illustrates those steps of the communication protocol that are important for the programmer, although most of these operations can be hidden, as described below. Consider user process 1 sending a message to user process 2. Step 1 indicates that user process 1 has written values into the shared memory array A, or has marshaled a message into array A. User process 1 initializes the send operation by sending a send request via the pipe to kernel process 1 (2). Kernel process 1 transfers the message to kernel process 2 (3), while both user processes can continue their computation. At some point during program execution (4), user process 1 synchronizes the pending send call by sending a *sync* request to kernel process 1. User process 1 blocks until kernel process 1 has signaled that it has delivered the message completely to the TCP protocol layer. At some time, user process 2 signals a receive request to kernel process 2 (5). Two cases are distinguished here. First, if this request has been issued after the message transfer has been started, kernel process 2 allocates buffer space to store the message. Buffer space is allocated according to the size of a message fragment (max. 4 Kbyte), and eventually linked into lists as indicated by array C. When user process 2 sends a *sync* request to the kernel (6), the message has to be copied into the requested memory region. Second, to avoid this copy, the receive request (5) should be transferred to kernel process 2, before the message arrives. In this

case, the message can directly be stored in user space (array B). The *sync* request of user process 2 (6) blocks until the entire message is stored in B. Finally, step 7 indicates, that user process 2 can access the message in the requested memory region.

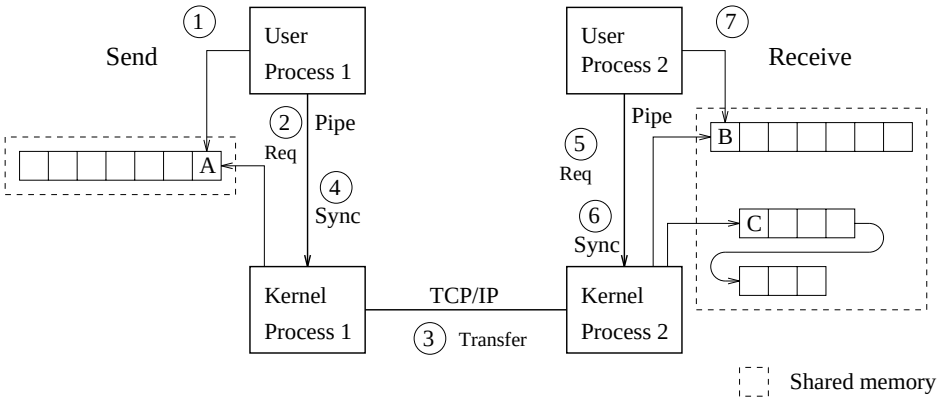


Figure 17 Data exchange via shared memory (cf. Fig. 14).

The program fragment corresponding to Fig. 15 can be given now. Both user processes initiate the receive before the send, to ensure that the arriving message is stored in array `bcol` without copying. Then, both processes initiate the transfer of the artificial boundary array `abcol`. The transfer of the messages is hidden behind the calculation, or, interpreted differently, idle times of the kernel process are utilized for calculation. The synchronization of both send and receive operations by means of `wait` calls completes the example. It is in the programmer's responsibility to guarantee that the `initread` call is issued before the message arrives, if additional copying shall be avoided.

```
Calculate artificial boundary
rid = initread(s, bcol, len);
wid = initwrite(s, abcol, len);                                    (I.3)
Calculate inner domain
wait(wid);
wait(rid);
```

The idea of splitting the send and receive primitives into two calls is not new. It has been introduced to utilize hardware support for latency hiding. Intel's NX asynchronous message passing primitives `isend` and `irecv` together with `msgwait` are an example [18]. The `request` and `release` communication primitives for the Seamless architecture provide a similar notion of latency tolerance, yet based on more sophisticated hardware support [12].

4 Experimental Results

In this section the overhead induced by the process splitting is analyzed. Also, measurements of application performance are given that illustrate the benefit of latency hiding in the Internet. To demonstrate the communication performance of systems typically available today, we give performance measures of three different networks in Fig. 18. LANs are typically connected via Ethernet, or sometimes via FDDI technology. For large-scale distributed computing, the heterogeneous Internet is available.

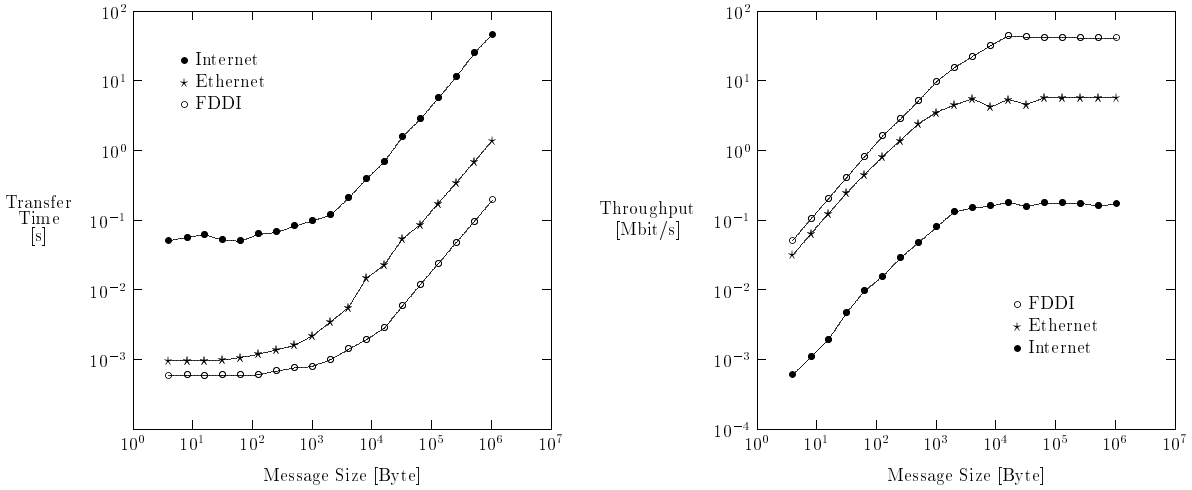


Figure 18. Performance comparison of different networks.

The data given in Fig. 18 have been collected with ping-pong measurements via stream sockets without message buffering. Comparing the latencies with those of a typical supercomputer, that today are in the order of a microsecond, the presented networks are about three to four orders of magnitude slower.

4.1 Overhead Analysis

Equation (4) expresses the sensitivity of communication latency hiding gain with respect to the overhead induced by the latency hiding implementation. The following first order effects contribute to this overhead: Context switches between user and kernel process, pipe communication for synchronization and TCP protocol overhead for managing the actual message passing. The major sources of overhead are the four pipe communications each accompanied by a context switch of the steps 2, 4, 5 and 6 in Fig. 17. Table 2 lists measurements of *user space-to-user space* messages with 8 byte of user data, transferred via direct stream socket connections and without message copying (direct), and transferred by the latency hiding protocol (lathide). For convenience, the overhead of four pipe communications, including the context switch, are included according to Table 1. The measurements are conducted on two machines of the listed model connected by an Ethernet.

Machine Model Pair	Overhead [ms]	direct [ms]	lathide [ms]
IBM RS6000/590	0.2	0.596	0.977
Sun SPARCstation1+	1.52	0.962	2.489
Sun SPARCstation10	0.44	0.442	0.996

Table 2. Overhead of latency hiding protocol on the Ethernet.

The sum of direct transfer times and overhead due to pipe communication and context switches matches the measured transfer times with the latency hiding protocol fairly well. If the receiving kernel process can copy the message directly into user space, the effect of overhead on message transfer times decreases with larger message sizes, because the overhead is a fixed number. Figure 19 shows the *penalty factor*, which is the ratio of transfer times with latency hiding and those of direct stream socket transfer.

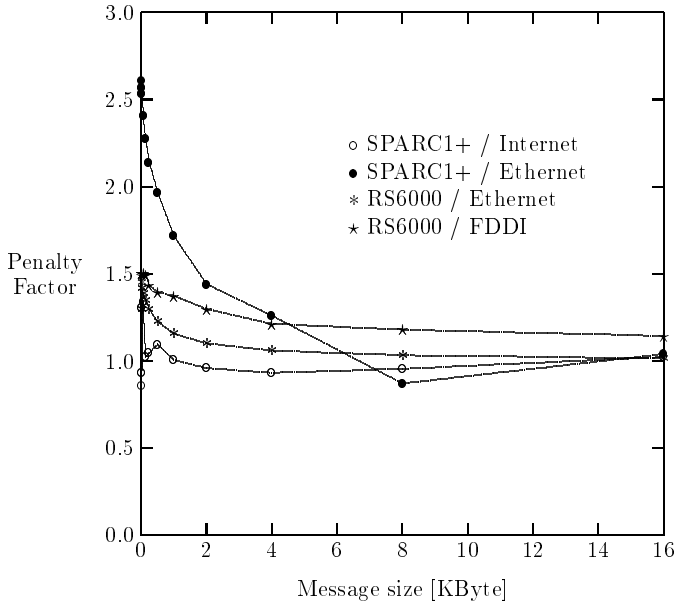
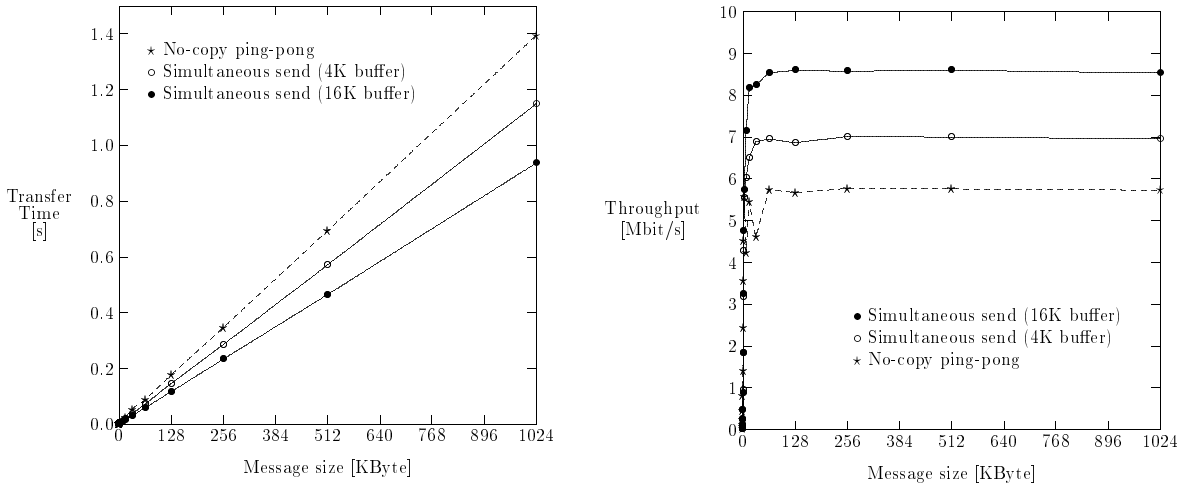


Figure 19. Penalty of latency hiding.

In the Internet with startup times of about $50\,ms$ the latency hiding overhead of about $1\,ms$ is negligible. In the Ethernet, the overhead is negligible for messages sizes bigger than about 4 Kbyte. The overhead increases relative to the network speed, as the comparison of the Ethernet and FDDI penalties show for identical machines (RS6000). It is obvious that faster hardware mechanisms for synchronizing user and kernel process are desirable to reduce the impact of overhead especially for small messages on the Ethernet. The penalty factor is approximately of the size of the gain for small messages. If these messages can be hidden, the penalty can be tolerated.

The hardware impacts the overhead term Ω of the model above. Workstations usually operate with an additional network interface chip, that allows for concurrent communication and calculation. Memory bandwidth will contribute to the overhead, if CPU and network interface compete for access. More important are the software sources of overhead though. Synchronization, including context switching, has been analyzed already. These costs also include socket abstraction overhead and TCP/IP protocol overhead. Queueing pending messages in the kernel and message fragmentation are minor overheads caused by the latency hiding implementation. Furthermore, the application structure might require marshaling messages to avoid the even higher cost of sending multiple messages.

An important advantage of the developed protocol is the possibility to simultaneously send off messages between two processes before receiving them. In contrast to the above ping-pong benchmarks, this communication pattern can deadlock with pure stream socket communication. The interleaved transmission and receipt of message fragments of the protocol makes much better use of the network bandwidth and achieves almost peek transfer rates.



(a) (b)

Figure 20. Performance of simultaneous send via Ethernet.

The results of simultaneous communication in Fig. 20 have been measured between two SPARCstation1+ connected via an Ethernet. Message sizes correspond to the amount of user data. The figures compare the cost of ping-pong styled communication via stream sockets with simultaneous send through the communication kernel, with default socket buffer size (4 Kbyte) and increased buffer size of 16 Kbyte. The performance hardly improves for send and receive buffer sizes bigger than 16 Kbyte. Thus, for this particular configuration, 16 Kbyte buffers are well matching the hardware granularity, i.e. the overhead of the protocol and the raw network transfer times. For larger message sizes, the

transfer of a message is in the average about 50% faster with simultaneous send through the communication kernel than with a ping-pong styled message exchange via direct stream connections. This result favors the use of programming models that run a single program on different processors in parallel (such as SPMD).

4.2 Two-Processor Experiments

Illustrating the gain of the UNIX latency hiding implementation is fairly complex because of implementation details of the TCP protocol. Recalling the example of Fig. 1, and its parallel computation by means of algorithms $(A.1)$ and $(A.2)$, the speedup of algorithm $(A.2)$ with respect to $(A.1)$ has to be attributed to different mechanisms, depending on the socket buffer size. Two cases have to be distinguished: (1) The send buffer size is not smaller than the message size, and (2) the send buffer size is smaller than the message size. In the former case, the UNIX send system call returns immediately after copying the entire message into the send buffer. Latency hiding is implemented on the hardware level, where the physical separation of CPU and network interface hardware guarantees partial overlapping of calculation and communication. Latency hiding overhead is induced by the TCP and IP protocols that require CPU cycles. In the second case, the UNIX send system call would block until the first part of the message is transferred and the second part is copied into the send buffer, as described in 3.1.1 and illustrated in Fig. 13. Because the UNIX send call is issued by the communication process, the user process utilizes those CPU cycles during which the blocking send operation would otherwise wait for TCP events. These wait times can be largely attributed to the protocol overhead at the corresponding peer process.

To illustrate the difference between these two cases, two experiments have been performed on a two-processor configuration that investigate the effect of software granularity on gain G. The explicit finite difference solver has been run on a rectangular two-dimensional domain, with the versions $(I.1)$, $(I.2)$, and $(I.3)$, implemented on top of the developed protocol. The behavior of the blocking send system call in $(I.2)$ has been simulated by executing the synchronizing `wait` calls of $(I.3)$ immediately after communication has been initialized. The three versions were executed with two different grid sizes in the y direction. In experiment (a), the y direction contained 500 grid points, corresponding to 500 doubles or 4000 byte per message. Experiment (b) was conducted with 1000 grid points in the y direction, yielding 8000 byte messages. Using the default socket buffer size of 4 Kbyte for the send and receive buffers, the message size in experiment (a) is smaller than the buffer size, corresponding to case (1). In experiment (b) it is bigger, which corresponds to case (2). The number of time steps is $T = 50$ in all experiments.

These experiments yield the gains of implementation $(I.2)$ versus $(I.1)$ (blocking send) and $(I.3)$ versus $(I.1)$ (latency hiding). Figure 21 shows gains versus a normalized domain

size or software granularity, Fig. 21(a) for experiment (a), and Fig. 21(b) for experiment (b). Software granularity is varied by changing the mesh size in the x-dimension. Thus, the number of instructions is varied, while communication volume is kept constant. $\gamma_s = 1$ corresponds to the domain size of 5 grid-points on the x-axis per processor. Accordingly, $\gamma_s = 120$ denotes an x-range of 600 points per processor.

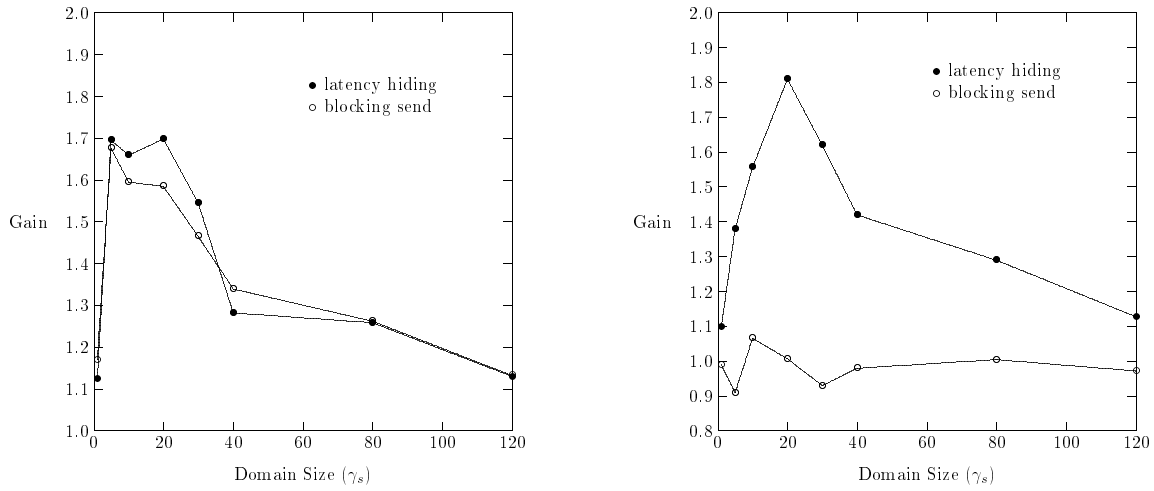


Figure 21. Two-processor gain on the Internet.

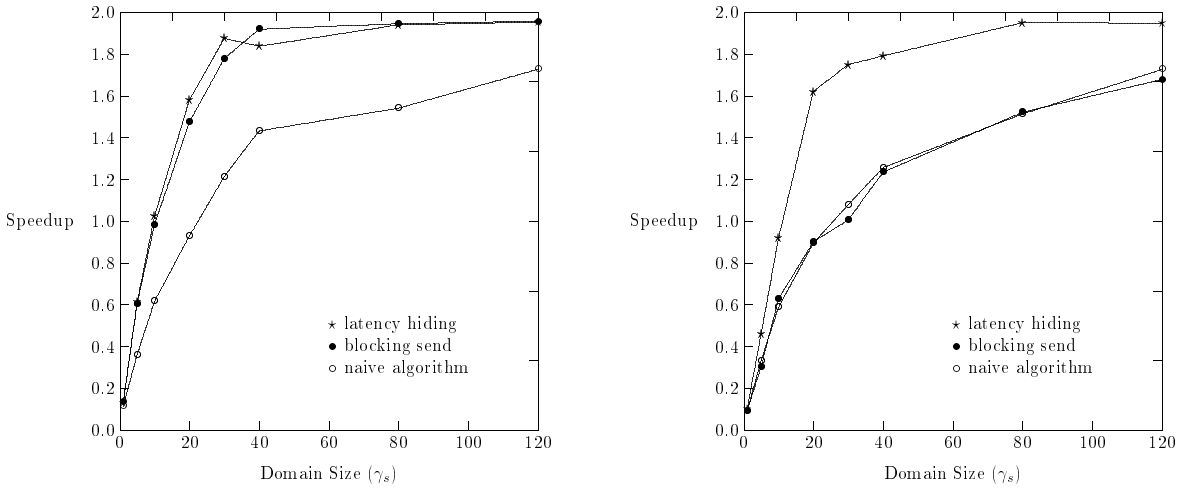


Figure 22. Two-processor speedup on the Internet.

These measurements were performed with one SPARCstation1+ at ETH, Zürich, connected with a SPARCstation2 at MIT, Cambridge, via the Internet. The different computational power of these machines is irrelevant, because the application enforces tight

synchronization, and the configuration behaves like two SPARCstation1+ [7]. The difference between Fig. 21(a) and Fig. 21(b) visualizes the relation between message size and buffer sizes. In Fig. 21(a), the message size is smaller than the send buffer size. Because the network interface handles this case almost independently of the CPU, both the blocking send and the latency hiding version perform similar (within the measurement accuracy of the Internet). In Fig. 21(b), the blocking send requires the message to be transferred in two steps, as described in Section 3.1.1. Here, only the latency hiding protocol delivers a gain. If the message size increases, the gain can be expected to be even higher. Figure 22 shows the corresponding speedup curves of the three measurements per experiment. The reference for calculating the gain values are the speedup values of implementation ($I.1$) (naive algorithm).

In the previous experiments, the dependency of message size and the socket buffer sizes on latency hiding performance became obvious. To illustrate the effect of latency hiding for large messages, and different communication implementations, another suite of experiments has been performed between two SPARCstation10 on an Ethernet. The finite difference solver was run with 100,000 grid points in the y-direction, corresponding to 800,000 byte per message. This size exceeds the maximal socket buffer size by far.

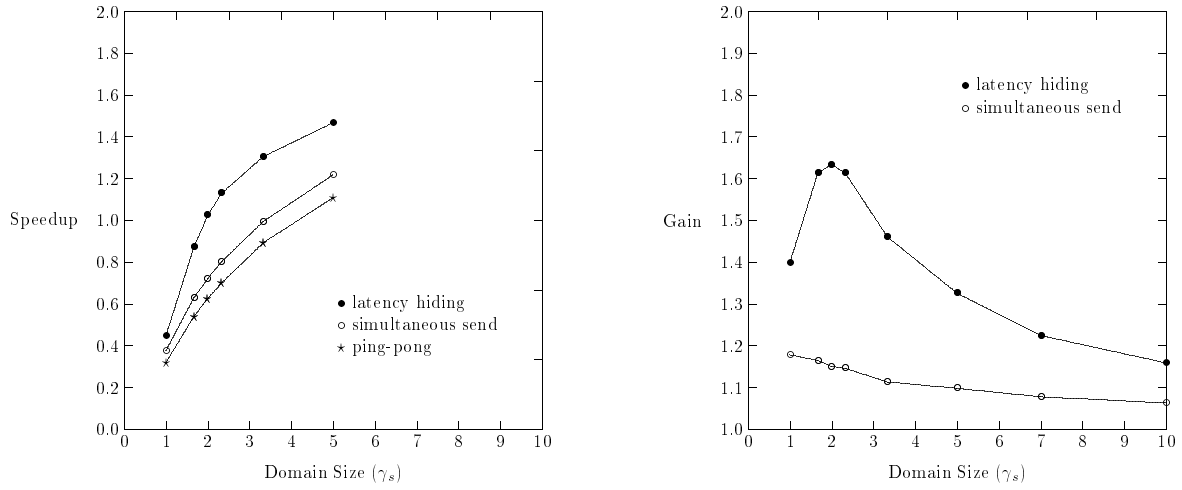


Figure 23. Two-processor speedup and gain on the Ethernet.

The reference implementation (ping-pong) operates with pure stream socket communication. To avoid deadlock, this version is programmed with ping-pong styled communication, i.e. two programs are required, one that first sends a message and then receives, and the other that first receives a message and then sends. To illustrate the performance gain of the developed protocol without latency hiding but simultaneous sending of messages, version

($I.1$) has been implemented (simultaneous send). In contrast to pure stream socket communication, the processes do not deadlock but achieve higher network utilization according to Fig. 20. The third version is the latency hiding implementation (latency hiding). For both versions with simultaneous send and latency hiding, the communication kernel has been tuned by switching on the TCP_NODELAY flag and setting the socket buffer sizes to 32 Kbyte. The resulting speedup and gain values are presented in Fig. 23.

The domain size of $3 \times 100,000$ grid points per processor corresponds to $\gamma_s = 1$. The huge communication volume demonstrates the performance capabilities of the Ethernet, although it is clearly an unrealistic partitioning of the problem domain. For $\gamma_s \geq 5$, the memory requirements for the sequential measurements exceeded the physical memory. Consequently, swapping activities instead of latency hiding lead to large speedups of values > 10, and are therefore omitted in Fig. 23(a).

Compared to the Internet experiments, the calculation of the artificial boundary cannot be neglected, especially for smaller software granularities. Thus, the latency hiding is less ideal, which results in a lower maximal gain. Furthermore, the hardware granularity of the Ethernet configuration is different from the Internet configuration, probably smaller due to the faster network despite of the faster machines. Therefore, the influence of the overhead of the latency hiding implementation increases relatively, which also decreases the maximum of gain G.

4.3 Multi-Processor Experiment

To further illustrate the validity of our latency hiding model for multiprocessor systems, we compare with the results of a performance model of two Krylov subspace algorithms, with and without latency hiding. In [24], de Sturler and van der Vorst gave the following runtime model for their conjugate gradient algorithms without latency hiding

$$T(p) = (9 + 4n_z)t_{fl}\frac{N}{p} + 6(t_s + 3t_w)\sqrt{p},$$

and with latency hiding

$$T_{lh}(p) = (9 + 2n_z)t_{fl}\frac{N}{p} + \max\Big(2n_z t_{fl}\frac{N}{p} + 24(t_s + 3t_w),\ 6(t_s + 3t_w)\sqrt{p}\Big)$$

for a mesh-based multi-processor, which has been validated by experiments. For the evaluation of gain, the values of all variables in these equations are kept constant with exception of the number of processors p. The equations can therefore be written:

$$T(p) = \frac{a}{p} + b\sqrt{p},$$
$$T_{lh}(p) = (1 - f)\frac{a}{p} + \max\Big(f\frac{a}{p} + \omega,\ b\sqrt{p}\Big).$$

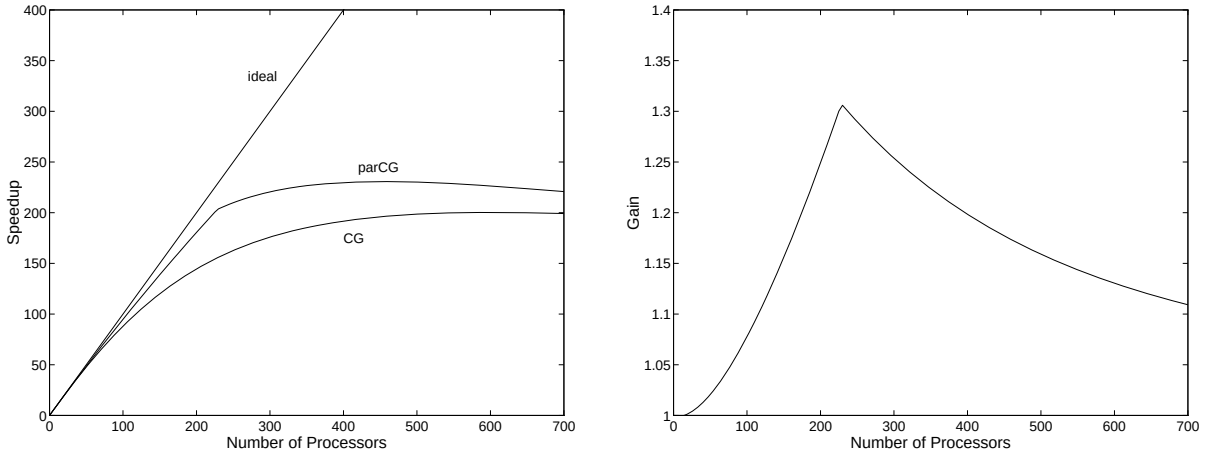
Figure 24(a) shows the speedups $S(p) = T(1)/T(p)$ (CG) and $S_{lh}(p) = T_{lh}(1)/T_{lh}(p)$ (parCG), as presented in [24]. Based on the equality $G = T(p)/T_{lh}(p)$, the gain can be calculated, and reduces to:

$$G = \frac{1 + p\sqrt{p}\; b/a}{(1-f) + max\Big(f + \omega p/a,\; p\sqrt{p}\; b/a\Big)}.$$

Comparing this formula, derived from runtime modeling of two conjugate gradient algorithms, with our general formula (6) based on granularities, it turns out that the communication volume is $q = \sqrt{p}$, and the ratio of granularities λ and overhead Ω are:

$$\lambda = a/b = \frac{(9+4n_z)t_{fl}N}{6(t_s + 3T_w)}, \qquad \Omega = \omega p/a = \frac{24(t_s + 3T_w)}{(9+4n_z)t_{fl}N}p.$$

Hence, this algorithmic runtime model can be expressed by the latency hiding model, and the benefit of communication latency hiding can be illustrated by the notion of gain, as shown in Fig. 9.3.



(a) (b)

Figure 24. Speedup and gain of Conjugate Gradient implementation.

This example elucidates the impact of communication latency hiding in multi-processor systems. The maximal gain $G = 1.307$ is obtained with $p = 229$ processors. The speedup without latency hiding and 229 processors is $S(229) = 155.7$. With latency hiding, the G-fold speedup is $S_{lh}(229) = 203.5$, which is a substantial improvement. The maximum speedup of the version without latency hiding is 200.2 with 601 processors. With latency hiding, the maximum speedup is 230.6 with only 453 processors, despite the fact that the overhead of this latency hiding implementation is relatively high.

5 Conclusions

The notion of communication latency hiding gain as the ratio of speedups of an algorithm with and without latency hiding, and a model of gain was introduced that expresses the gain in terms of software granularity, hardware granularity and overhead. The model captures the opportunities and limits of latency hiding by overlapping communication and calculation. Communication latency hiding is especially valuable for parallel computing in computer networks where relatively large messages cause high communication cost. Many algorithms are directly suited or can be restructured to allow for overlapping communication and calculation.

The analysis of latency hiding motivated the introduction of a protocol layer for efficient and deadlock-free message passing in UNIX environments. This protocol is implemented by means of a separate process, called communication kernel. Providing latency hiding is not the only feature enabled by this process separation. First of all, the kernel is message-driven, that is, it reacts on each incoming message depending on a message tag. This allows not only for user level message exchange, but various actions at runtime system level. The separation provides flexibility with respect to various communication abstractions. Message queueing, selective message passing, remote procedure calls, and so on can be implemented. Building these mechanisms on top of the communication kernel offers a machine independent interface, which allows for the development of portable runtime systems for parallel languages.

The validity of the communication latency hiding model has been illustrated by presenting experimental data and a model from numerical algorithm design that both match the model. The experimental data showed the validity of latency hiding for distributed parallel computing. In particular, for appropriate applications, latency hiding can make the Internet appear as suited for solving large problems with parallel computing as a supercomputer. Furthermore, communication latency hiding impacts massively parallel processing, because efficiency can be improved considerably, and, accordingly, the number of employable processors can be increased. As a side effect, measurements indicated that faster synchronization and context switching methods are desirable to reduce the overhead that especially elongates the transfer of small messages. Implementations based on light-weight processes may perform better.

Acknowledgement

I thank Walter Gander for providing me with a stimulating working environment and proof reading. Tom Casavant supported and contributed to this work in an unrivalled manner. He became the backbone of this work. The comments of Peter Arbenz improved the

quality of this report substantially. I enjoyed to share my experience and to discuss various related issues with Edouard Bugnion, Kevin Gates, and Christoph Sprenger. Thanks also to Charles Leiserson for his confidence to provide us with an account at MIT.

References

[1] MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, April 1994. (Distribution: `netlib`)

[2] Arbenz P., Lüthi H.P., Mertz J.E., Scott W., "Applied Distributed Supercomputing in Homogeneous Networks," *International Journal of High Speed Computing* **4**(2), 1992, 87–108.

[3] Arbenz P., Sprenger C., Lüthi H.P., Vogel S., *SCIDDLE: A Tool for Large Scale Distributed Computing*, Technical Report **213**, Departement Informatik, ETH Zürich, March 1994.

[4] Atallah M.J., Black C.L., Marinescu D.C., Siegel H.J., Casavant T.L., "Model and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations," *Journal of Parallel and Distributed Computing* **16**(4), 1992, 319–327.

[5] Bala V., Bruck J. et al., "The IBM External User Interface for Scalable Parallel Systems," *Parallel Computing* **20**(4), 1994, 445-462.

[6] Bertsekas D.P., Tsitsiklis J.N., *Parallel and Distributed Computation: Numerical Methods* (Prentice-Hall, Englewood Cliffs, 1989).

[7] Cap C.H., Strumpen V., "Efficient Parallel Computing in Distributed Workstation Environments," *Parallel Computing* **19**(11), 1993, 1221–1234.

[8] Carter J.B., Zwaenepoel W., "Optimistic Implementation of Bulk Data Transfer Protocols," *Performance Evaluation Review* **17**(1), 1989, 61–69.

[9] Comer D.E., Stevens D.L., *Internetworking with TCP/IP, Vol II: Design, Implementation, and Internals* (Prentice-Hall, Englewood Cliffs, 1991).

[10] Crowcroft J., Wakeman I., Wang Z., Sirovica D., "Is Layering Harmful?," *IEEE Network Magazine* **6**(1), 1992, 20–24.

[11] Eicken von T., Culler D.E., Goldstein S.C., Schauser K.E., "Active Messages: A Mechanism for Integrated Communication and Computation," In *Proceedings of the 19th International Symposium on Computer Architecture*, ACM Press, Gold Coast, Australia, May , 1992.

[12] Fineberg S.A., Casavant T.L., Pease B.H., "Hardware Support for the Seamless Programming Model," In *The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, October, 1992, 353–360.

[13] Griebel M., Huber W., Störkuhl T., Zenger C., "On the Parallel Solution of 3D PDEs on a Network of Workstations and on Vector Computers," In *Computer Architecture:*

Theory, Hardware, Software, Applications, Springer-Verlag, 1993, 276–291.

[14] Gropp W., "Parallel Computing and Domain Decomposition," In *5th International Symposium on Domain Decomposition Methods for Partial Differential Equations*, SIAM, Norfolk, May 6–8, 1991, 349–361.

[15] Hwang K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability* (McGraw-Hill, New York, 1993).

[16] Leffler S.J., McKusick M.K., Karels M.J., Quarterman J.S., *The Design and Implementation of the 4.3BSD UNIX Operating System* (Addison-Wesley, Reading, 1989).

[17] Nakanishi H., Rego V., Sunderam V., "Superconcurrent Simulation of Polymer Chains on Heterogeneous Networks," in *Supercomputing '92*, Minneapolis, IEEE, 1992, 561–569.

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

[19] G.D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods* (Oxford University Press, New York, 1985).

[20] Stevens R.W., *UNIX Network Programming* (Prentice Hall, Englewood Cliffs, NJ, 1990).

[21] Stone H.S., *High-Performance Computer Architecture* (Addison-Wesley, Reading, 1993).

[22] Strumpen V., "A Large-Scale Metacomputer Approach for Distributed Parallel Computing," In *High-Performance Computing and Networking*, Vol. II, Springer-Verlag, Munich, April 18–20, 1994, 278–285.

[23] Sturler de E., *A Performance Model for Krylov Subspace Methods on Mesh-based Parallel Computers*, Technical Report, Swiss Scientific Computing Center, to appear.

[24] Sturler de E., Vorst van der H.A., "Communication Cost Reduction for Krylov Methods on Parallel Computers," In *High-Performance Computing and Networking*, Vol. II, Springer-Verlag, Munich, April 18–20, 1994, 190–195.

[25] Sunderam V.S., "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience* **2**(4) 1990, 315–339.

[26] Wunderlich R., Hege H.-C., Grammel M., *On the Impact of Communication Latencies on Distributed Sparse LU Factorization*, Technical Report **SC 93-28**, Konrad Zuse-Zentrum für Informationstechnik Berlin, 1993.