Analysis and Optimization of Software Pipeline Performance on MIMD Parallel Computers

Rob F. Van der Wijngaart*, Sekhar R. Sarukkai†, Pankaj Mehra‡

Abstract

Observations show that fine-grain software pipelines on MIMD parallel computers with asynchronous communication suffer from dynamic load imbalances which cause delays in addition to the expected pipeline fill time. An analytical model that explains these load imbalances is presented. Optimization derived from the analysis leads to significant improvements in program performance. The results of applying this optimization to general pipeline algorithms on the Intel iPSC/860, Intel Paragon and IBM SP/2, as well as to pipelined tri-diagonal equation solvers on the Intel Paragon and the IBM SP/2, are presented.

1 Introduction

Software pipelines are among the most popular methods to extract parallelism from numerical processes that exhibit recurrences. For example, the most widely-used parallel flow solver at NASA Ames, POVERFLOW developed by Weeratunga [8], employs pipelines in the Gaussian elimination phase of the Beam-Warming algorithm in order to resolve data dependencies and maintain a reasonable load balance.

It is well-known that software pipeline performance concerns a trade-off between amount of computation performed between successive communications and the number of communications; the latter may be decreased (through grouping) at the expense of a longer pipeline fill time. In the context of this paper, a pipeline is said to be fine-grained if the amount of time spent in performing work between successive communications is of the same order of magnitude as the communication time itself. Since processor speeds are generally increasing faster than network bandwidths, it is envisioned that fine-grain pipelines will become more important to study, as medium-grain pipeline applications move toward the fine end of the granularity spectrum. For example, what may constitute a medium-grain pipeline on an older machine such as an Intel iPSC/860 may well be

*MCAT Inc.; currently with MRJ Technology Solutions, NASA Ames Research Center, Moffett Field
†Recom Technologies; currently at Hewlett-Packard Laboratories, Palo Alto
‡Recom Technologies; currently at Tandem Computer, Santa Clara

considered a fine-grain pipeline on the high-end IBM SP/2. Fine granularity can also be encountered in simple computational kernels in very long pipelines, where grouping would severely degrade the load balance.

Pipeline operations have been well studied for a number of years (see for example Dubey and King *et al.* [3, 5]) in various contexts. In this paper we specifically concentrate on modeling software pipelines implemented on message-passing distributed-memory machines. Our desire for modeling these pipelines is motivated by the need to explain significant differences in modeled and observed execution times of a number of applications that use software pipelines as a means of exploiting parallelism. The observations garnered from visual depictions of program execution expose unexpected non-linear processor delays that we set forth to study analytically. We investigate the structure of the delays based on a simple parallel performance model. The model successfully tracks the non-linear behavior of pipelines and proves to be an effective tool for estimating overall performance of applications with pipelines. Building on the results of our investigation, we also obtain an optimal strategy for reducing the processor delays. The proposed model and optimizations are experimentally verified on three different parallel machines: the Intel iPSC/860, Intel Paragon, and IBM SP/2. We find that execution times as predicted by the model compare favorably to actually measured performance.

A number of techniques such as aggregating messages and increasing granularity (grouping) are routinely used to optimize the performance of pipelines, as discussed, for example, by Hiranandani and Palermo *et al.* [4, 7]. Here we demonstrate that in addition to the above optimizations, a significant part of the delay of fine-grain software pipelines implemented on MIMD distributed-memory parallel computers can be eliminated by removing dynamic load imbalances created by interrupts. As shown in Section 5, a number of currently available machines are amenable to this optimization. In fact, results indicate that optimizations can lead to a reduction in execution time by up to a factor of four on the SP/2 for very-fine-grained pipelines.

The optimization strategy is also employed to optimize the pipelined solution of a large set of independent linear tri-diagonal equations on the Intel Paragon and IBM SP/2. Such sets of equations arise naturally from the implementation of the Alternating Direction Implicit method (ADI), of which the Beam-Warming algorithm is an example. Our results show improvements of around 40% using the derived optimization strategy on the Paragon and of more than 60% on the IBM SP/2.

2 Observations and previous work

In order to describe previous and the current work on software pipelines, we consider a model problem consisting of N identical independent computational tasks, to be completed by p identical processors. Each task is divided uniformly into p ordered subtasks. Every j^{th} subtask within each task is assigned to processor number j. A data dependence exists among the subtasks within a task; subtask j cannot start before subtask $j-1$

has finished. The pipeline algorithm is constructed as follows. Processor 1 completes its first subtask and sends a message to processor 2, indicating that its first subtask can commence. Subsequently, processor 1 completes its second subtask while processor 2 completes its first. After both subtasks are completed, processor 1 sends another message to 2, and processor 2 signals processor 3. This pattern is repeated during the so-called pipeline start-up, and after $p-1$ subtasks have been completed by processor 1, all processors are active and the pipeline is filled, provided that $N \geq p$.

An intuitively appealing description of this pipeline process assumes that each subtask requires a fixed, constant amount of wall clock time, and that a network-borne message latency creates a delay between issuance of a message on a processor and reception of that message on its successor in the pipeline. This simple model—which we will call *traditional*—results in a perfectly synchronized pipeline operation in which no processor ever waits for data once the first message is received; this in turn leads to pipeline fill times and total completion times that are linear functions of the processor number p.

We use the performance-monitoring tool AIMS described by Yan *et al.* [11] (Automated Instrumentation and Monitoring System) to illustrate qualitatively the pipeline behavior of a problem where $p = 4$ and $N = 100$, implemented on an Intel iPSC/860 hypercube computer. In Figure 1, horizontal striped bars indicate processor status. Dark

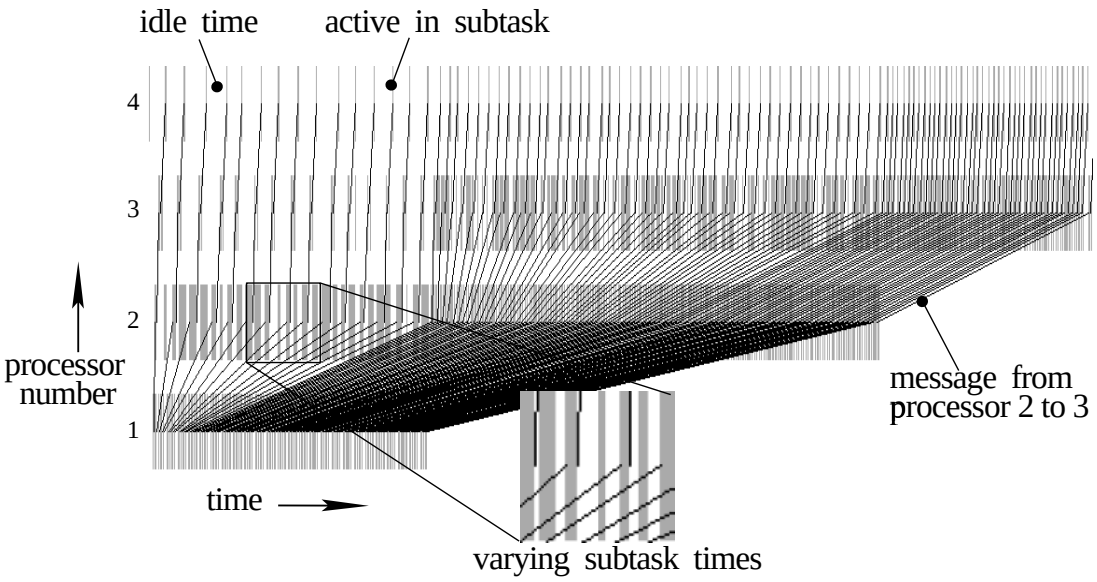


Figure 1: AIMS processor activity status for pipeline algorithm

sections signify that a processor has started a subtask. White space within a bar indicates that a processor is not doing any computational work, but is sending a message or waiting for one instead. Black lines connecting bars denote messages being passed among processors. Message lines in the AIMS time line originate where the sender blocks

(suspends program execution), waiting for local completion of communication, and terminate where the receiver unblocks (resumes program execution), having received the message. Therefore, the end of a message line should not be confused with the instant when the message was actually delivered to the receiving processor. Although the amount of computational work per subtask is constant, the amount of wall clock time spent within subtasks varies (see inset), as does the amount of time spent waiting between subtasks. This variation shows because AIMS does not explicitly monitor system-level operations on each processor.

A clear fan-out of message transfer lines is visible between processor 1 and processor 2 (which was also observed by Palermo [7]), and to a lesser extent between 2 and 3. This implies that some subtasks take longer to complete on a certain processor than on its predecessor in the pipeline. But there are also phases in the pipeline algorithm during which message transfer lines are parallel, indicating that communicating subtasks of successive processors in the pipeline take equal amounts of time.

Obviously, total completion time is not linear in p, and not all subtasks take equal amounts of wall clock time, since not all message transfer lines are parallel. Moreover, in contrast with previous studies (*e.g.* King [5]) that assume that messages always arrive before the receive has been posted so that processors never need to wait for data, AIMS probes show that in many instances processors are actually waiting for messages from their predecessors in the pipeline.

A more realistic model such as that by Adve *et al.* [1] ascribes certain communication overheads to the processors in the parallel machine, rather than to the communication network. More specifically, they assume that a fixed amount of cpu time is spent by a processor that receives and processes (*'handles'*) a message. Since the first processor never receives a message, this model can account for a disparity in completion times between the first and the second processor, but not for nonlinearities in completion time on higher-numbered processors. Introduction of an additional overhead incurred by the processor that sends a messages does not change the model qualitatively.

Perhaps the most relevant work is that which models pipeline stalls in the context of hardware pipelines. Dubey [3] presents a bubble propagation model for hardware pipeline performance, using delay distributions of individual stages. That analytical model is geared towards estimating performance of short pipelines (5 to 6 stages). While Dubey discusses issues specifically relevant in the context of hardware pipelines with different amounts of work for pipeline stages, our approach is geared towards predicting the performance of software pipelines with arbitrarily many stages and a uniform computational work distribution, and towards the study of the effect of interrupts on successive stages.

3 Performance analysis

Here we postulate a communication model that completely explains the observed pipeline behavior, and that offers a means for optimization described in Section 4. Its most impor-

tant feature is the occurrence of interrupt events that generate dynamic load imbalances.

Assumptions:

1. The message length is zero (similar to assuming infinite network bandwidth). This is a reasonable approximation for fine-grained pipelines, where messages are usually very short and communications are latency-dominated. It also leads to somewhat simpler algebra. Later this restriction will be lifted.

2. The computational work associated with each subtask requires a constant period of c cpu time units. Later we will allow c to be changed on the first processor in the pipeline.

3. When a message is *sent* by a processor, a constant *non-overlappable* send overhead of s time units is incurred immediately by that processor. For short messages a significant part of the send overhead may be due to the construction of the path to the receiving node. This has been observed on the Intel iPSC/860. Nearest-neighbor communication assures that s is indeed constant.

4. When a message *arrives* at a processor, a constant *non-overlappable* receive-interrupt overhead of r_i time units is incurred immediately by that processor.

5. When a message is *used* by a processor, a constant *non-overlappable* receive-handling overhead of r_h time units is incurred by that processor.

Assumption 4 is what distinguishes our model most prominently from previous models, such as those presented by Adve and Palermo [1, 7], who use s and r_h, but not r_i. In the context of real-time parallel computing, Mraz [6] presents data to show how operating system support functions hinder a parallel application's ability to respond to messages in a timely manner. The effects of interrupts on the variance of point-to-point communications in parallel programs are considered, but they are not modeled or analyzed. Moreover, in Mraz' study the interrupts originate from a source other than the application program itself, whereas we examine interrupts generated by the pipeline program only. Interrupts generate dynamic load imbalances because they consume cpu time during intermediate pipeline stages. Certain communication protocols (CMMD, MPI) may actually support withholding messages until a request for them has been posted in order to avoid copying of message buffers (see, for example, Saphir [9]). This kind of communication delay automatically provides the type of pipeline optimization described in Section 4, but at the cost of explicit synchronization between all pipeline segments (see Section 5).

In general, each processor experiences four pipeline phases, identified schematically in Figure 2. They are discussed below in chronological order.

Blocking. The processor is blocked while waiting for a message signaling that its first subtask can be started (pipeline fill). Processor 1 does not exhibit this phase.

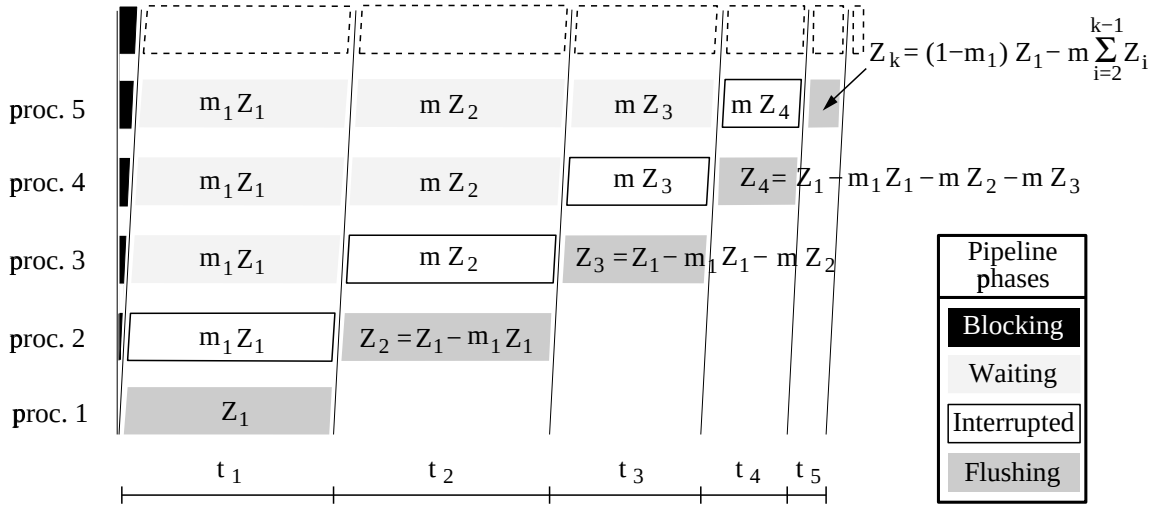


Figure 2: Four pipeline processor phases

WAITING. This phase is dominated by waiting for messages to arrive from the predecessor. The corresponding set of subtasks on the predecessor takes a long time to complete, either because that processor is in the wait phase itself, or because it is being interrupted frequently; incoming and outgoing message transfer lines are parallel, which means that subtasks take equally long on successive processors in the pipeline. Processors 1 and 2 do not exhibit this phase.

INTERRUPTED. The processor experiences long aggregate subtask completion times because it is interrupted at high frequency by its predecessor. The predecessor is in the flush phase, and sends messages rapidly. Processor 1 does not exhibit the interrupt phase.

FLUSHING. The processor is not interrupted by its predecessor in the pipeline, either because there is no predecessor (processor 1), or because the predecessor has already completed all its subtasks and has no more messages to be issued. However, all processors in this phase still need to handle already arrived messages (except processor 1), perform the remaining subtasks, and send messages to their successor (except the last processor); incoming message transfer lines are parallel, since the preceding processor has fired messages from the same no-interrupt state, so subtasks take equally long (except on processor 2).

We now analyze in detail the durations of the different phases on processor k ($k \geq 2$). The phases can be grouped into two major modes, the blocked mode and the active mode. The blocked mode coincides with the blocking phase; no subtasks can be started yet due to the pipeline fill. The active mode contains the remaining three phases. Its total length is the sum of all subtask durations. In the sequel, subtask durations are taken to be the

average of all such durations within a specific phase, *including all communication and system overheads*.

Subtasks can be grouped into three classes of equal subtask lengths. These correspond to the periods t_1, t_2 through t_{k-1}, and t_k, respectively, which are indicated in Figure 2. Periods are defined recursively; they equal the amount of time needed to finish all the subtasks whose messages have been received (*i.e.* whose receive calls have been cleared) during the corresponding period of the predecessor. If there is no such corresponding period, then the period equals the time needed to flush all remaining subtasks.

The number of subtasks executed during the flushing period on processor k is Z_k. Clearly, $Z_1 = N$, since processor 1 has only a single period, during which all subtasks are flushed. The number of subtasks executed by processor 2 during period t_1 is $m_1 Z_1$. m_1 is the ratio of subtask duration on processor 1 over that on 2 during the first period. Its inverse $1/m_1$ is the frequency with which processor 2 gets interrupted by 1 (its units are interrupts per subtask). This frequency may be fractional, since it is an average over many subtasks. It is determined as follows. A single subtask on processor 2 during t_1 is interrupted by an average of $1/m_1$ messages from processor 1. Hence, it lasts $c + s + r_h + r_i/m_1$ time units. Since every subtask on processor 1 issues exactly one message, it takes $(c + s)/m_1$ time units for that processor to generate the $1/m_1$ interrupts of processor 2. Equating the two lapses yields

$$c + s + r_h + r_i/m_1 = (c + s)/m_1 \tag{1}$$

so that we find

$$m_1 = \frac{c + s - r_i}{c + s + r_h}. \tag{2}$$

Note that the number of subtasks executed during t_1 by all processors numbered higher than 2 is $m_1 Z_1$, just as on processor 2; they are wait-dominated due to the slowdown of the frequently interrupted processor 2.

The interrupt frequency $1/m$ during flushes other than in period t_1 is slightly lower, because the subtasks on the flushing processor last longer than those on processor 1. A flushing processor is no longer interrupted by its predecessor, but it does need to handle the already arrived messages, which incurs an overhead of r_h time units per subtask. So now we equate the subtask duration $c + s + r_h + r_i/m$ on the interrupted processor to $(c + s + r_h)/m$ time units on the interrupter, and obtain [1]

$$m = \frac{c + s + r_h - r_i}{c + s + r_h}. \tag{3}$$

[1] Eq. (2) implies that $r_i < c + s$. The meaning of violation of this inequality is that processor 2 will be interrupted constantly, until processor 1 exhausts all its subtasks. In this case the parallel pipeline breaks down, and (partial) serial execution results. If $c + s < r_i < c + s + r_h$, then m_1 is zero and the first processor finishes completely before the second has finished even one subtask, but all subsequent processors are properly pipelined. If $c + s + r_h < r_i$, then both m_1 and m are zero, and execution is completely serial. Although this is possible in principle, it is not of interest for this analysis.

Notice that the interrupt frequency $1/m$ is the same for all flushing periods other than that on processor 1, as the sender states and receiver states are the same for all subsequent nodes. So the number of subtasks executed on the interrupted and waiting processors during t_j equals mZ_j, where $j \geq 2$.

Evidently, each processor ultimately has to execute all Z_1 subtasks, so the number of subtasks Z_k remaining during the flushing phase on processor k is

$$Z_k = Z_1 - m_1 Z_1 - m \sum_{j=2}^{k-1} Z_j \,, \quad \text{with} \quad Z_1 = N \,. \tag{4}$$

This equation constitutes a simple recursion, which is most easily solved by computing $Z_{k+1} - Z_k$. It immediately follows that $Z_{k+1} = (1-m)Z_k$, so

$$Z_k = N(1-m_1)(1-m)^{k-2} \,, \quad k \geq 2 \,. \tag{5}$$

The total duration T_k^a of the active mode on processor k is now computed as

$$\begin{aligned} T_k^a &= t_1 + \sum_{j=2}^{k-1} t_j + t_k \\ &= (c+s)Z_1 + (c+s+r_h+r_i/m)m \sum_{j=2}^{k-1} Z_j + (c+s+r_h)Z_k \\ &= N(c+s) + N(1-m_1)\left[c+s+r_h+\left(1-(1-m)^{k-2}\right) r_i/m\right] \,, \quad k \geq 2 \,. \end{aligned} \tag{6}$$

The total duration T_k^b of the blocked mode on processor k is determined as follows. We assume that the first subtask of any processor is interrupted exactly once, so that

$$T_k^b = T_{k-1}^b + c + s + r_i + r_h \,. \tag{7}$$

This may not be true for $k = 2$, because processor 1 can potentially send multiple messages arriving during the execution of the first subtask on processor 2. But the error that follows from this simplifying assumption is small, and results only in a uniform shift of the time lines of all subsequent processors, if any. Since T_1^b is zero, we find:

$$T_k^b = (k-1)(c+s+r_i+r_h) \,. \tag{8}$$

Summing the durations of the idle and active modes, we finally obtain for the total pipeline duration T_k on node k:

$$\begin{aligned} T_k \stackrel{\text{def}}{=}\ & T_k^b + T_k^a = (k-1)(c+s+r_h+r_i) + N(c+s) + \\ & N(1-m_1)\left[c+s+r_h+\left(1-(1-m)^{k-2}\right) r_i/m\right] \,, \quad k \geq 2 \,. \end{aligned} \tag{9}$$

Substituting Eqs. (3) and (2) into (9) and defining the two relative interrupt and handling overheads

$$\alpha = \frac{r_h}{c+s} \,, \quad \beta = \frac{r_i}{c+s} \,, \tag{10}$$

we obtain the scaled delay:

$$\delta \mathbf{t}_k \stackrel{\text{def}}{=} \frac{T_k}{N(c+s)} - 1 = (1+\alpha+\beta)\frac{k-1}{N} + \frac{\alpha+\beta}{1+\alpha-\beta}\left\{1+\alpha-\beta\left(\frac{\beta}{1+\alpha}\right)^{k-2}\right\}. \quad (11)$$

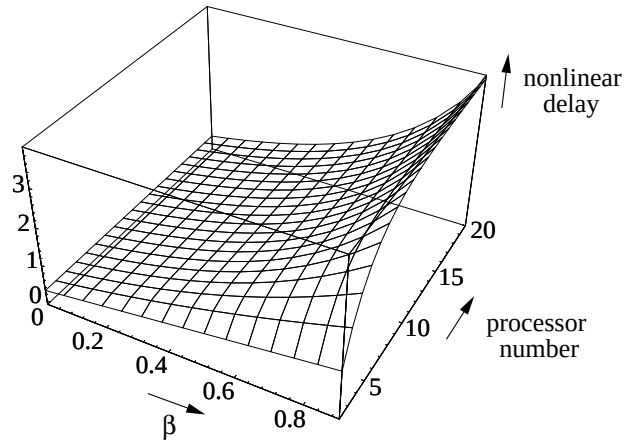


Figure 3: Scaled nonlinear delay for $\alpha = 0.3$

The first term on the right hand side of Eq. (11) signifies the expected scaled completion delay due to pipelining for processor k without nonlinear interrupt effects. The second term is the additional nonlinear delay due to the wave-like propagation of the effects of message interrupt and handling overheads in the interrupt states (Figure 2). This nonlinear delay is depicted in Figure 3 for a fixed scaled handling overhead of 0.3, and for a range of scaled interrupt overheads.

For large k or small β, the asymptotic scaled delay is:

$$\delta \mathbf{t}_k = (1+\alpha+\beta)\frac{k-1}{N} + \frac{(\alpha+\beta)(1+\alpha)}{1+\alpha-\beta}. \quad (12)$$

This expression is not valid for the last node of the pipeline, since no more send operations are required, but the error is negligible for pipelines involving many processors [2].

Since the subtask durations for the last processor, p, in the waiting phase are determined by processor $p-1$, the number of subtasks performed during t_1 through t_{p-2} is the same as before. But the fan-out in the interrupt phase will be reduced, and actually

[2]If the number of processors is small, then the timing for the last processor may differ significantly from the asymptotic value, or even from Eq. (11). This situation occurs often with pipelines resulting from three-dimensional domain decompositions (see, for example, Van der Wijngaart [10]). The number of processors in a pipeline in this case is the cube root of the total number of processors, which is usually a small number.

completely vanishes if $s \geq r_i$; in the latter case $t_p = 0$, and $T_p = T_{p-1} + (c + s + r_i + r_h)$, so that

$$\begin{aligned}
\delta\mathbf{t}_p &= \frac{T_{p-1} + (c + s + r_i + r_h)}{N(c+s)} - 1 \\
&= \delta\mathbf{t}_{p-1} + \frac{1 + \alpha + \beta}{N} \\
&= (1 + \alpha + \beta)\frac{p-1}{N} + \frac{\alpha + \beta}{1 + \alpha - \beta}\left\{1 + \alpha - \beta\left(\frac{\beta}{1+\alpha}\right)^{p-3}\right\}, \quad s \geq r_i\,. \qquad (13)
\end{aligned}$$

If $s < r_i$, then there will still be a fan-out, and $t_p \neq 0$. Rewriting Eq. (1) for node p, we obtain:

$$r_i/m_p + r_h + c = (r_h + c + s)/m_p\,, \qquad (14)$$

so that

$$m_p = \frac{c + s + r_h - r_i}{c + r_h}\,. \qquad (15)$$

Modifying Z_p accordingly (see Eq. (5)), we obtain for the number of subtasks remaining during the flushing phase:

$$Z_p = N(1 - m_1)(1 - m_p)(1 - m)^{p-3}\,. \qquad (16)$$

The total time spent on processor p is now easily calculated as:

$$T_p = T_{p-1} + (c + s + r_h + r_i) + Z_p(c + r_h)\,. \qquad (17)$$

Introducing a third relative overhead:

$$\gamma = \frac{s}{c+s}\,, \qquad (18)$$

we find the following scaled delay:

$$\begin{aligned}
\delta\mathbf{t}_p &= (1 + \alpha + \beta)\frac{p-1}{N} + \\
&\quad \frac{\alpha + \beta}{1 + \alpha - \beta}\left\{1 + \alpha - \left(\frac{\beta}{1+\alpha}\right)^{p-3}\left[\beta - \frac{(\beta - \gamma)(1 + \alpha - \beta)}{1 + \alpha}\right]\right\}, \; s < r_i\,. \qquad (19)
\end{aligned}$$

Eqs. (13) and (19) can be combined to form:

$$\begin{aligned}
\delta\mathbf{t}_p &= (1 + \alpha + \beta)\frac{p-1}{N} + \\
&\quad \frac{\alpha + \beta}{1 + \alpha - \beta}\left\{1 + \alpha - \left(\frac{\beta}{1+\alpha}\right)^{p-3}\left[\beta - \max(0, \beta - \gamma)\frac{1 + \alpha - \beta}{1 + \alpha}\right]\right\}. \qquad (20)
\end{aligned}$$

3.1 Finite message length

If the message length is nonnegligible and the communication bandwidth between nodes is finite, the model has to be changed slightly; the message send overhead may increase if data gets copied locally, but this can be absorbed in the definition of s, since it adds to the overhead on the sending processor. Similarly, copy costs on the receiving end can be incorporated in r_h, and longer processor tie-up with interrupts results in larger r_i. If the network is not autonomous, the sending processor needs to participate in the entire transfer of the message, which can again be absorbed in the definition of s; in that case the model as presented above remains the same. If the network is autonomous, the sending processor can resume subtask execution once the message is placed on the network. Here the finite bandwidth results in an increased blocking time. If we assume a constant message bandwidth of b bytes per second and a constant message size of B bytes between subtasks, then the message transfer time q per message on a contention-free network is:

$$q = \frac{B}{b}\,. \tag{21}$$

Consequently, the block time becomes:

$$T_k^b = (k-1)(c+s+q+r_i+r_h) \tag{22}$$

The finite message size has no effect on the rest of the phases, since the *frequency* of departing and arriving messages on an autonomous network is not affected by the linear shift q. Introducing the scaled message travel time

$$\sigma = \frac{q}{c+s}\,, \tag{23}$$

we find in general that

$$\delta\mathbf{t}_k = (1+\sigma+\alpha+\beta)\frac{k-1}{N} + \frac{\alpha+\beta}{1+\alpha-\beta}\left\{1+\alpha-\beta\left(\frac{\beta}{1+\alpha}\right)^{k-2}\right\}\,, \tag{24}$$

and for the last processor in the pipeline:

$$\begin{aligned}\delta\mathbf{t}_p \;=\; & (1+\sigma+\alpha+\beta)\frac{p-1}{N} + \\ & \frac{\alpha+\beta}{1+\alpha-\beta}\left\{1+\alpha-\left(\frac{\beta}{1+\alpha}\right)^{p-3}\left[\beta-\max(0,\beta-\gamma)\frac{1+\alpha-\beta}{1+\alpha}\right]\right\}\,.\end{aligned} \tag{25}$$

4 Optimization

It follows from the analysis in Section 3 that the reason for the nonlinear slowdown of the pipeline algorithm is the high frequency of interruption of certain sets of subtasks. All delays are incurred during the interrupt phase, whose fan-out propagates as a wave through the pipeline. No such wave pattern would be observed if the *first* processor were to send messages at a lower rate. This suggests that the pipeline algorithm can be optimized by artificially increasing the amount of work performed by the first processor for each subtask. Another type of optimization can be obtained if the computational cost of a subtask is not fixed, but is manipulated by grouping several (independent) subtasks together. This coarsening of the granularity reduces the number of messages—and hence the communication overhead—at the expense of increased blocking time. Moreover, the relative overheads α and β decrease, reducing the nonlinear slowdown as well.

4.1 Optimal padding

We replace c by $c' + c$ on processor 1, where c' is a padding amount, and redo the analysis, keeping the computational work for the subtasks on other processors the same. We use the symbol $'$ to indicate perturbations due to padding. Note that no benefits can be obtained by increasing the padding beyond the sum of handling and interrupt overheads, since at that point processor 2 is forced to wait for data from processor 1. Consequently,

$$0 \le c' \le r_i + r_h \,. \tag{26}$$

If we assume that the padding operations within each subtask on processor 1 take place *after* each send operation, then the processor block times stay uniformly the same as before, *i.e.*,

$$\left(T_k^b\right)' = T_k^b \,. \tag{27}$$

In addition, we find that

$$t_1' = N(c' + c + s) \,. \tag{28}$$

Expressions for the numbers of subtasks per period stay the same, but the definition of m_1 changes:

$$m_1' = \frac{c' + c + s - r_i}{c + s + r_h} \,. \tag{29}$$

The total time spent on node k becomes:

$$\begin{aligned} T_k' \;=\; & (k - 1 + N)(c + s) + (k - 1)(r_i + r_h) + Nc' + \\ & N(1 - m_1)\left[c + s + r_h + \left(1 - (1 - m)^{k-2}\right) r_i/m\right] \,, \quad k \ge 2 \,. \end{aligned} \tag{30}$$

Introducing a scaled padding:

$$\tau = \frac{c'}{c+s}\,, \tag{31}$$

we obtain the following expression for the scaled completion delay (cf. Eq. (11)):

$$\begin{aligned}\delta\mathbf{t}'_k &= (1+\alpha+\beta)\frac{k-1}{N} + \tau + \frac{\alpha+\beta-\tau}{1+\alpha-\beta}\left\{1+\alpha-\beta\left(\frac{\beta}{1+\alpha}\right)^{k-2}\right\} \\ &= \delta\mathbf{t}_k + \tau\frac{\beta}{1+\alpha-\beta}\left\{\left(\frac{\beta}{1+\alpha}\right)^{k-2} - 1\right\}\,. \end{aligned} \tag{32}$$

As expected, for $k = 2$ (the second processor in the pipeline) we find that $\delta\mathbf{t}'_k = \delta\mathbf{t}_k$, independent of the value of τ; different amounts of padding cause differences in the durations of the interrupt and flush phases, respectively, but the sum of these times will be the same, since processor 2 never needs to wait for data from processor 1. This implies that optimization of the pipeline can never reduce the increase in execution time that processor 2 incurs over processor 1, and savings can only be obtained starting with processor 3. Eq. (32) represents a linear function in τ, whose extremal values are attained at the boundaries of the interval defined by Eq. (26). As before, we will assume that $\beta < 1$ (*i.e.* $r_i < c+s$), so the coefficient of τ is negative. Consequently, the total delay is minimized by maximizing τ, yielding:

$$\tau = \alpha+\beta \quad \text{or} \quad c' = r_i + r_h\,. \tag{33}$$

Note that the optimal τ is independent of the processor number k and the amount of work per subtask c. Optimal padding results in the equality

$$m'_1 = 1\,, \tag{34}$$

which implies that subtasks on processor 1 now take equally long as on 2. In fact, all subtasks are now performed within the waiting phase for all processors, and the other phases (except for the initial pipeline blocking) are totally eliminated. Since the optimal τ does not depend on the number of processors, the suggested padding strategy is globally optimal; no padding of subtasks on other processors can improve the completion time. We finally find:

$$\delta\mathbf{t}_p^{opt} = (1+\alpha+\beta)\frac{p-1}{N} + \alpha + \beta\,. \tag{35}$$

This expression has the appearance of a synchronized pipeline result with a slightly modified subtask duration. In the case of finite-message-length communications on an autonomous network, the expected correction term σ appears, but the optimal amount of padding remains the same:

$$\delta\mathbf{t}_p^{opt} = (1+\sigma+\alpha+\beta)\frac{p-1}{N} + \alpha + \beta\,. \tag{36}$$

4.2 Optimal grain size

Padding by itself is usually not the best strategy for optimizing fine-grain pipelines. Grouping several subtasks together to increase granularity (as in Hiranandani and Palermo [4, 7]), if possible, is generally more effective, because it reduces the communication overhead *and* the frequency of interrupts. But grouping and padding can also be combined to obtain an optimally performing pipeline. This is especially effective if the optimal grain size is still relatively fine, which may be the case for long pipelines and/or very fast processors.

The total amount of computational work for each processor is $N\,c$, as before. But now instead of dividing this into N subtasks of size c each, we allow for grouping into $N^{opt} = N/n^{opt}$ subtasks of size $c^{opt} = n^{opt}\,c$. Again, we minimize the effect of the wave-like delay propagation, so the optimal padding of the subtasks on the first processor in the pipeline keeps the same form, and the final completion time for node p is as in Eq. (36).

Some assumptions about the impact of message and subtask consolidation have to be made. Most notably, if copying of message buffers takes place on the sending and receiving processors, then the send and receive handling overheads have to be functions of the subtask grouping size n. A simple linear model is adopted in which copying and computing speeds and network bandwidth are constants. A processor interrupt is assumed to take a fixed amount of time. The following substitutions are made in Eq. (36):

$$\begin{aligned} N &\leftarrow N/n \\ c &\leftarrow nc \\ s &\leftarrow s + s_1 nB \\ r_i &\leftarrow r_i \\ q &\leftarrow nB/b \\ r_h &\leftarrow r_h + r_1 nB \end{aligned}$$

This leads to the following final completion time:

$$\begin{aligned} T_p &= N/n\,\{nc + s + s_1 nB + r_h + r_1 nB + r_i\} + \\ &\quad \{nc + s + s_1 nB + r_h + r_1 nB + r_i + nB/b\}\,(p-1) \end{aligned} \tag{37}$$

The optimal subtask grouping size is obtained by setting

$$\frac{\partial T_p}{\partial n} = 0\,, \tag{38}$$

which yields:

$$n^{opt} = \sqrt{\left(\frac{N}{p-1}\right)\left(\frac{s + r_h + r_i}{c + (s_1 + r_1 + 1/b)B}\right)}\,. \tag{39}$$

This result is equivalent to that obtained by Palermo ([7], Eq. (3)), provided that the variable *ovhd* is set equal to $s + r_h + r_i$, and that s_1, r_1 and b are incorporated in the variable *rate*. The corresponding optimal padding per subtask is:

$$c' = (r_i + r_h)/n^{opt} + r_1 B\,. \tag{40}$$

It follows from the last equation that padding vanishes in the limit of large grouping sizes if no copying of message buffers on the receiving end is performed.

5 Model verification and numerical application

In order to verify the validity of our performance model we use a synthetic test program in which the amount of computational work per pipeline segment can be varied, and a tri-diagonal linear-equation-solver application. All optimizations presented are for a prescribed grain size; no grouping is applied to reduce communication overheads.

5.1 Performance prediction of synthetic program

The synthetic test program is run on the Intel iPCS/860, and on the Intel Paragon and IBM SP/2, using the NX and MPI message passing libraries, respectively. Both versions use blocking send and receive calls to pass empty messages (the message length will be varied later). The primary data set obtained consists of final completion times on all 16 nodes involved in the program execution. The C routine used to produce an entry of the data set on the iPSC/860 is printed below. The node numbering is such that only nearest-neighbor communication occurs, which eliminates contention. The number of pipeline tasks executed is kept fixed at 256 (*i.e.* $N = 256$).

```
void pipeline(work, padding) int work, padding;
{ int          s, p, my_segment, next, first, last;
  double       *in, *out, q;

  my_segment = ginv(mynode()); next = gray(my_segment+1);
  first      = 0;              last  = numnodes() - 1;

  if (my_segment != first) for (p = 0; p < 256; p++)
        {crecv(p, in, 0);
        for (s = 0; s < 10*work; s++) q = 1.0/(s + 1.0);
        if (my_segment != last) csend(p, out, 0, next, 0);}
  else for (p = 0; p < 256; p++)
        {for (s = 0; s < 10 * (work+padding); s++) q = 1.0/(s + 1.0);
        csend(p, out, 0, next, 0);}
}
```

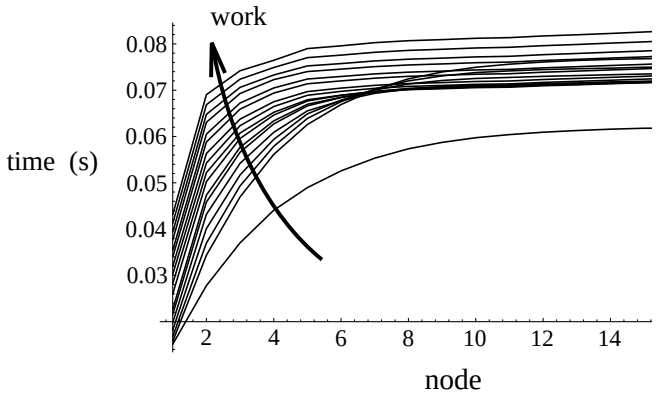


Figure 4: Completion times for increasing arithmetic loads on iPSC/860

The variable *work* determines the amount of arithmetic work for all pipeline segments, whereas *padding* determines the additional work per segment for the first node. A multiplication factor of 10 is used to scale the work to measurable amounts. The main program is a double loop over the *pipeline* routine for $work = 0$ **step** 1 **to** 16, and $padding = 0$ **step** 1 **to** 32, and computes ensemble averages for each parameter pair over 20 independent runs.

Figure 4 shows the final completion time versus node number for zero padding and varying amounts of work per pipeline segment (higher curve generally means more work per segment). Interestingly, the completion time is not a monotonically increasing function of the work per segment; this also follows from the the theoretical model. From the data displayed in Figure 4 we compute the interrupt and handling overheads as follows. The first node provides the value of the scale factor $c + s$, since all it does is compute and send. From Eq. (9) and the identity $T_1 = N(c + s)$ (no padding), we derive that

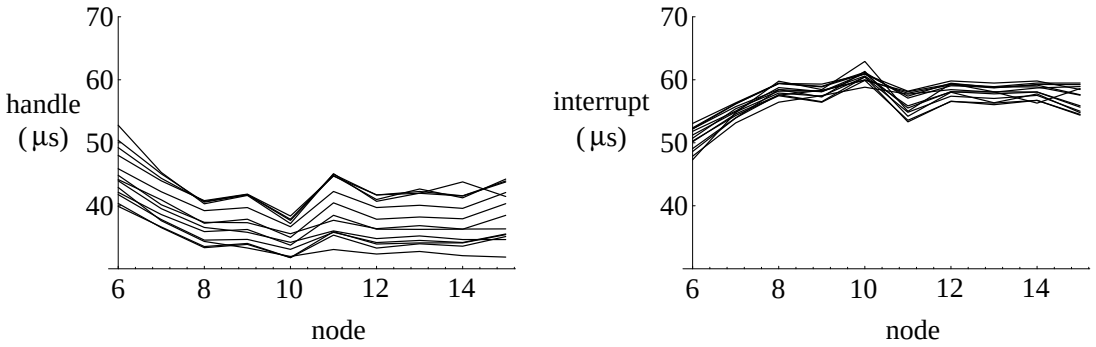


Figure 5: Measured receive overheads on iPSC/860

$r_i + r_h = T_2/(N + 1) - T_1/N$, which can be combined with Eq. (12) for larger values of k and $k + 1$ to compute α and β (and hence r_i and r_h) separately. The results of these

computations are depicted in Figure 5. Again, different curves pertain to executions with different amounts of work per pipeline segment, starting with $work = 4$. The first few curves with very small amounts of work per pipeline segment ($work < 4$) are left out, since the overhead of the extra test (i.e. **if** (my_segment ! = last)) in the case of receiving processors introduces significant skew in comparison to the work performed. It can also be seen that the overhead results are rather noisy for the small node numbers. By averaging the results of these computations for node 15, we obtain the values of $r_h = 39\mu$s and $r_i = 57\mu$s. Assuming that s is small, this result is well within the range of the measurements obtained by Bokhari [2] on the iPSC/860 installed at NASA Ames. He determined that the transfer time of an empty message to a neighboring node on the hypercube takes approximately 95μs.

Qualitative validity of the performance model derived above is demonstrated in Figure 6, which shows completion times on nodes 2 and 16, respectively, for the entire (*work,padding*) parameter space. As predicted by Eq. (32), the completion time on node

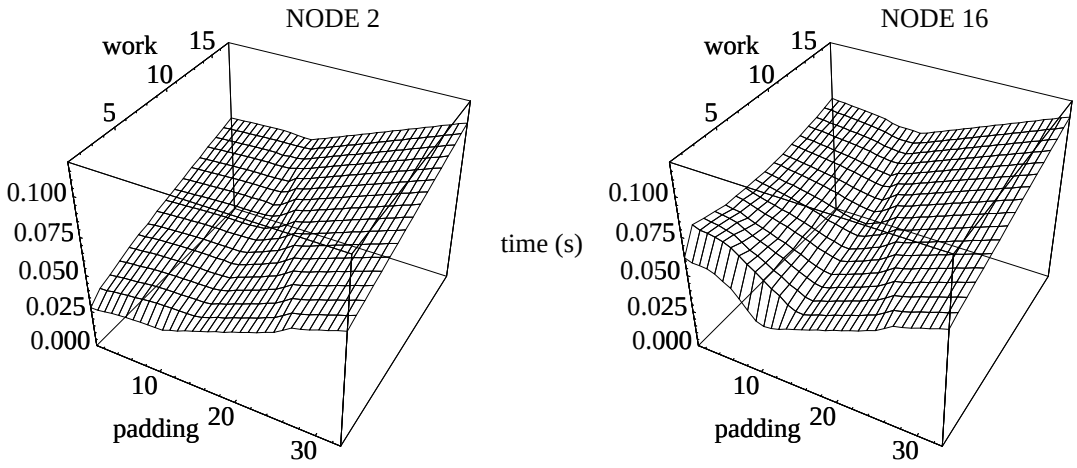


Figure 6: Completion times on the second and last pipeline nodes on iPSC/860

2 stays roughly constant while the padding is below the optimal amount. Completion times on node 16 show that optimal padding is a more or less fixed quantity ($padding = 13$), independent of the amount of computational work per pipeline segment.

5.1.1 Verification of basic model and optimization

In Figure 7 we show the actual cpu time involved with optimal padding (c'). The upper curve is obtained by subtracting the non-padded completion times from those at *padding* = 13 for the first node. The lower curve depicts the computed sum of handle and receive overheads. Theory predicts that the two curves be horizontal and coincident. Although they deviate and exhibit some scatter, they are close in absolute value.

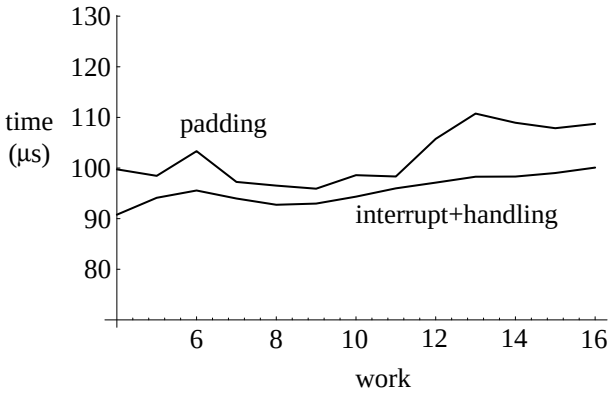


Figure 7: Comparison of receive overheads and optimal padding on iPSC/860

The achieved improvements in execution time for optimal padding versus no padding are plotted in Figure 8, alongside the theoretically predicted improvements based on the values of r_h, r_i and $c + s$ determined above. Note that the quantity 'work' (per pipeline segment) in this figure is now given in microseconds instead of in test program loop count, since the latter has no meaning for theoretical predictions. Agreement between predicted and measured speed-ups is quite good.

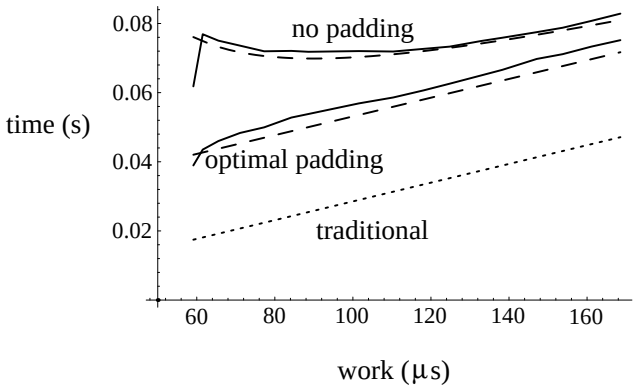


Figure 8: Predicted (dashed) and measured (solid) completion times with and without padding, and traditional linear model (dotted), on iPSC/860 (16 nodes)

In the same figure we also depict expected performance using traditional linear analysis (dotted line), which takes pipeline start-up time into account, but ignores the effect of dynamic load imbalance due to interrupts, *i.e.* $T_p = N(c + s) + (c + s + r_i + r_h)(p - 1)$. The traditional model, which assumes that all message receive costs are borne by the communication network, significantly underpredicts completion times, by up to a factor of three. Interestingly, most previous models [1, 3, 5, 7] underpredict the execution time as well, even in comparison with the optimally padded scheme.

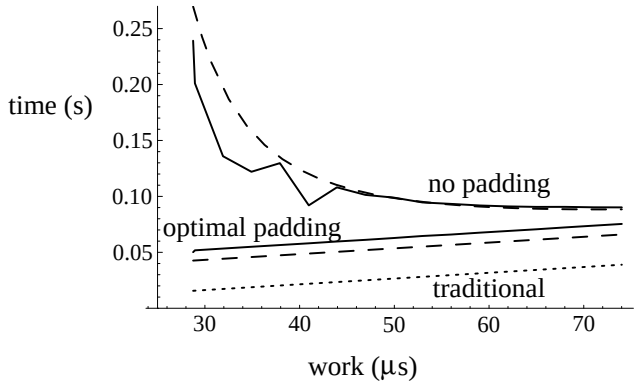


Figure 9: Predicted (dashed) and measured (solid) completion times with and without padding, and traditional linear model (dotted) on SP/2 (16 nodes)

The test program is also run on the IBM SP/2 using the MPI communication library and the same parameter set as before, and an increased number of pipeline stages ($N = 500$). Some minor modifications have to be made to avoid compiler optimizations that eliminate dead code. The size of the messages that are passed between successive pipeline stages is varied between 0 and 40 bytes, with no noticeable effect on final execution time. The resulting figures for r_h and r_i are 14μs and 40μs, respectively. In Figure 9 we compare the expected and measured performance of the pipeline procedure again. Now there is significant noise for small amounts of work per pipeline segment. This is due to the fact that the SP/2 processors and the switching network are not fully dedicated to the parallel job, but suffer from interference from system processes that may be started on a node during parallel program execution. This can sometimes cause exceptional delays and ruin the load balance. In order to limit the effects of such disturbances, we take the *minimum* execution time of 20 independent runs per parameter pair, rather than the average. However, when the first node in the pipeline is slowed down by a competing process, the result is the same as that of padding, and the resulting total execution time will be substantially less than expected. Other than these excursions, the model again performs quite well. It is important to note that the improvement obtained by padding is up to a factor of four for the finest-grain computations. This results from the increased floating point speed of the processor, which is not accompanied by a corresponding reduction in the interrupt-handling time. Apparently, the presence of message co-processors on the SP/2 does not alleviate the dramatic influence of interrupts on the receiving processor.

A similar result is found on the Intel Paragon, as is shown in Figure 10. The larger deviation of measured versus predicted execution times may be due to a larger influence of the message co-processors. The figures found for r_h and r_i are now $20\mu s$ and $58\mu s$, respectively.

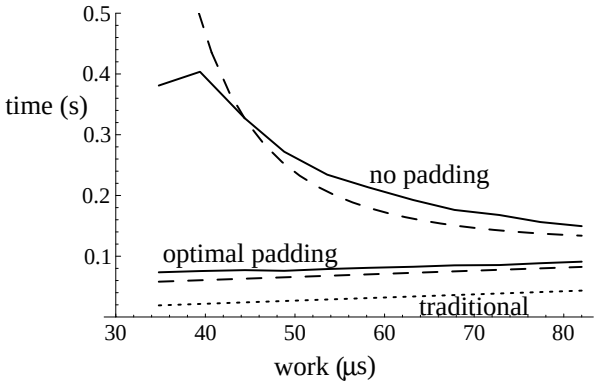


Figure 10: Predicted (dashed) and measured (solid) completion times with and without padding, and traditional linear model (dotted) on Paragon (16 nodes)

5.1.2 Comparison with other optimization strategies

Since the main focus of our investigations is fine-grained pipelines, we will assume the grain of the pipeline a fixed, small quantity, and will not consider grouping to optimize pipeline performance. However, some other possible optimization strategies should be investigated. So far, only blocking receives (*e.g.* mpi_recv) have been considered, which implies that each receive can only be posted after the previous one has been satisfied. A possible improvement may be obtained by utilizing non-blocking receive calls (*e.g.* mpi_Irecv), which allows posting of all receives before entering the pipeline loop.

Another potential improvement is offered by the explicit synchronization mechanism of mpi_Ssend, which does not allow the back-up of messages by a fast-firing processor in the flush stage; new messages may only be sent after all previous ones have been cleared by the receiving processor. The results of these potential optimizations on the SP/2 are presented in Figure 11 for $N = 256$.

Note that explicitly synchronized pipelines do not benefit from padding, so only the unpadded performance curve is presented for this communication mode. Obviously, explicit synchronization offers substantial improvement over both the blocking and the non-blocking communication modes for very fine-grained pipelines (small amounts of work per pipeline segments) if no padding is applied, but for slightly more coarse-grained pipelines the handshake overhead exceeds any possible gains. Synchronized communication compares even less favorably with optimally padded blocking or non-blocking receive modes, and is clearly not the right optimization strategy. For this relatively small number of pipeline stages and buffered receives (256), the non-blocking communication mode is superior to the blocking mode if no padding is applied. However, if optimal padding is applied the performance of both modes is equally good.

If, in addition, the number of pipeline stages is allowed to grow, the blocking communication mode outperforms the non-blocking mode with and without padding, with gains

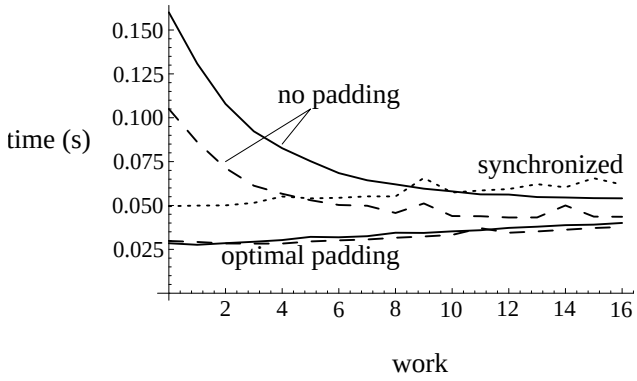


Figure 11: Completion times for blocking (solid), non-blocking (dashed), and synchronized (dotted) communication on SP/2 (16 nodes)

increasing with increasing N. Apparently, the cost of maintaining and scanning lists of buffered receive calls soon overwhelms any reduction in execution time due to not having to post receive calls during every pipeline segment. Figure 12 shows the difference in performance of the mpi_recv and mpi_Irecv communication protocols for a fixed amount of work per pipeline segment ($work = 3$) on the SP/2, with the number of pipeline stages ranging from 100 to 2000.

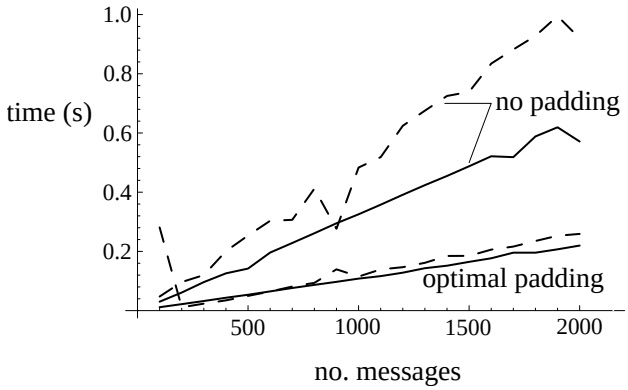


Figure 12: Completion times for blocking (solid) and non-blocking (dashed) communication on SP/2 for varying numbers of pipeline stages (16 nodes)

Finally we also compare the blocking, non-blocking and synchronized communication modes on the Intel Paragon, the results of which are presented in Figure 13, again for $N = 256$. Qualitatively, the performance of the various protocols is the same as on the SP/2, although the savings of the synchronized mode over the unoptimized blocking and non-blocking modes are substantially larger on the Paragon. Again, however, the optimally padded blocking communication mode offers the best overall performance.

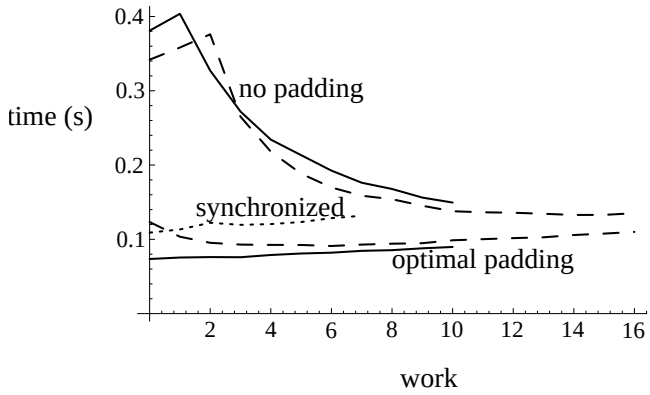


Figure 13: Completion times for blocking (solid), non-blocking (dashed), and synchronized (dotted) communication on Paragon (16 nodes)

5.2 Optimization of a pipelined tri-diagonal equation solver

We now apply padding to a problem taken from the solution of 3-dimensional partial differential equations (Van der Wijngaart [10]). Consider a cubical grid with 128^3 interior points, to be solved on a set of 512 nodes of an Intel Paragon XP/S without message co-processors. The interior of the grid is divided into $8^3 = 512$ blocks of 16^3 points, each of which is assigned to one processor. The linear systems resulting from the discretization of the 3-dimensional heat equation on the common 7-point stencil are solved using the Alternating Direction Implicit method (ADI). This means that in each coordinate direction a set of 128^2 independent tri-diagonal matrix equations of rank 128 has to be solved. In order to study this process we only need to consider 8 processors that contain a set of 16^2 complete grid lines of 128 points each. The most efficient serial algorithm for solving each equation is Gaussian elimination without pivoting. Since each processor contains only $1/8^{th}$ of each gridline passing through its grid block, this process is pipelined in order to obtain parallelism. We first consider the forward elimination, which requires 6 multiplications/additions and 1 division per grid point. Each node passes only three 8-byte numbers for each matrix equation to its successor. Figure 14 shows the AIMS performance profiles before and after padding, on the same time scale. Although the padding is slightly more than optimal (execution time on node 2 has increased), the performance gain is already around 40%. The reason why the optimization in this case could not be applied very accurately is that the monitoring process itself is intrusive and already provides some padding.

When the same experiment is performed on the IBM SP/2, we obtain a performance gain of 62% over the non-padded version for the forward elimination. As expected, the backsubstitution benefits more from optimization, since it is even finer-grained (only 2 multiplications/additions and no divisions per matrix row); padding leads to an execution time improvement of 67%. As an interesting detail, we point out that the unoptimized

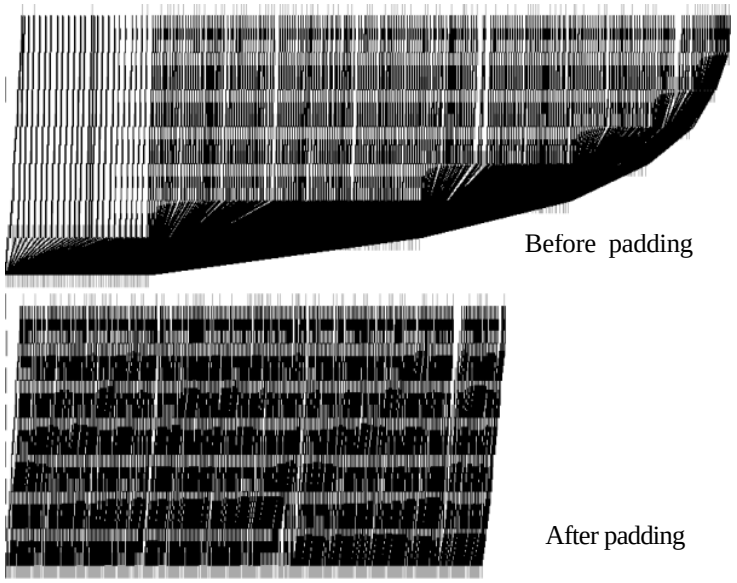


Figure 14: Optimization of tri-diagonal solver on Intel Paragon

forward elimination is actually faster than the unoptimized backsubstitution, despite that fact that more work gets done in the forward elimination. Optimizing through padding makes the backsubstitution faster again. It is worthy of note that optimal padding is independent of the type and amount of work done by the pipeline segments, as long as the cpu time spent in padding is the same, and equal to the sum of message handling plus interrupt overheads. This makes optimizing of pipelines through padding a particularly easy task.

These exercises also indicate potential for improvement of pipeline algorithms in production application programs, such as the parallel flow solver program by Weeratunga [8], which solves a collection of penta-diagonal matrix equations resulting from the discretization of the compressible Navier-Stokes equations. The method used is again ADI. As before, every processor receives a block of a structured grid, and is responsible for solving the part of each matrix system that corresponds to the grid points in the block. Obviously, this program can benefit from the optimization derived above, but it should also be noted that equal-sized grid blocks aggravate the dynamic load imbalance, since processors assigned blocks at a grid boundary have less work to do than others. Such processors are first in the pipeline during either the forward elimination or the backsubstitution, and so some additional padding is needed for an optimal algorithm.

6 Conclusions

We have investigated the performance of software pipelines on message passing architectures. Based on observations obtained using the performance monitoring and visualization tool AIMS, an analytical model for the pipeline behavior has been formulated. A salient feature of the model is the distinction between handling and interrupt overheads associated with receive operations. Four phases of pipeline operation are identified, each with different communication characteristics. The phases are analyzed independently, and predicted total completion time is obtained by summing the individual contributions. The most important phase is the *interrupt* phase, in which a processor has to suspend computations frequently to accommodate arriving messages issued by its predecessor in the pipeline. This causes a delay, which propagates to the other processors due to the pipeline nature of the computation. Since each stage in the pipeline adds a new delay to the previous one, the resulting total delay is a nonlinear function of the number of stages. Comparison of predicted and measured pipeline execution on several important parallel computer architectures—Intel iPSC/860, Intel Paragon, IBM SP/2—is favorable, and the modeled execution time follows the actual execution time accurately.

The model also points to possible optimization of the implementation of software pipelines in case grouping is not a suitable or feasible strategy. It as accomplished by reducing the frequency with which the first processor in the pipeline sends messages to its successor. This artificial slow-down actually improves overall program performance. The predicted magnitude of optimal first-processor retardation and the resulting reduction in execution time are again verified experimentally.

Other pipeline performance optimization strategies, such as explicit synchronization (*e.g.* mpi_Ssend), and non-blocking receive calls (*e.g.* mpi_Irecv) allowing posting of all receives in advance, are also investigated. These, however, are inferior to optimally padded blocking communications.

Speed-ups of around 1.7 on the Intel Paragon and of almost 3 on the IBM SP/2 are observed when applying padding to a pipeline version of a tri-diagonal equation solver application.

A remarkable feature of the proposed optimization is the ease of use of the optimization strategy on any architecture or application. Unlike many other optimizations, padding is independent of program structure; the optimal padding can either be directly applied by compilers that generate message-passing code, or can easily be hand-coded by the programmers.

Finally, it should be noted that the model also serves as an investigative tool for determining detailed communication characteristics of MIMD parallel computers, whose effects get magnified in software pipelines.

Acknowledgements

The authors would like to thank Rob Block from the University of Illinois for his early investigations and observations of software pipelines, and Bill Saphir from Computer Sciences Corporation for his many useful suggestions and insight, and for his insistence on dropping large amounts of extra redundant unneeded notation and equations. Maurice Yarrow of Sterling Software is due many thanks for his meticulous checking of all the arithmetic and logic in the paper, and for spotting some ugly mistakes.

References

[1] Adve, V., Mellor-Crummey, J., Reed, D. and Wang, J-C. Integrated performance analysis of data-parallel programs. *Proc. Workshop on Parallel Computing Systems*, LANL, Chatham, MA, 1994

[2] Bokhari, S.H. Complete exchange on the iPSC-860. NASA Tech. Rep. 91-4, ICASE, NASA Langley Research Center, Hampton, VA, 1991

[3] Dubey, P.K. and Flynn, M.J. A Bubble propagation model for pipeline performance. *J. Par. Distrib. Comput.* **23**, 1994.

[4] Hiranandani, S., Kennedy, K. and Tseng, C-W. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. *Proc. 1992 Int. Conf. Supercomput.*, ACM, Washington, DC, 1992

[5] King, C.-T., Chou, W.-H. and Ni, L.M. Pipelined data-parallel algorithms: Part I—Concept and modeling. *IEEE Trans. Par. and Distrib. Systems* **1**, 4, (October 1990), 470–485

[6] Mraz, R. *Reducing the variance of point-to-point transfers for parallel real-time programs*, IEEE Parallel and Distributed Technology, Winter 1994.

[7] Palermo, D.J., Su, E., Chandy, J.A. and Banerjee, P. Communication optimizations used in the PARADIGM compiler for distributed-memory multicomputers. *Proc. Int. Conf. Parallel Processing*, St. Charles, IL, 1994

[8] Ryan, J.S. and Weeratunga, S.K. Parallel computation of 3-D Navier-Stokes flowfields for supersonic vehicles. *Proc.* 31^{st} *Aerospace Sciences Meeting and Exhibit*, AIAA, Reno, NV, 1993, paper no. 93-0064

[9] Saphir, W.C. Message buffering and its effect on the communication performance of parallel computers. NASA Tech. Rep. RNS-94-004, NASA Ames Research Center, Moffett Field, CA, Apr. 1994

[10] Van der Wijngaart, R.F. Efficient implementation of a 3-dimensional ADI method on the iPSC/860. *Proc. Supercomput. '93*, IEEE Computer Society/ACM, Portland, OR, 1993, pp. 102-111

[11] Yan, J.C., Sarukkai, S.R., and Mehra, P. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs, *J. Software Practice and Experience* (April 1995)