NASA-CR-197758

(NASA-CR-197758) ALGORITHMS FOR PARALLEL FLOW SOLVERS ON MESSAGE PASSING ARCHITECTURES Final Report (MCAT Inst.) 32 p

N95-26588

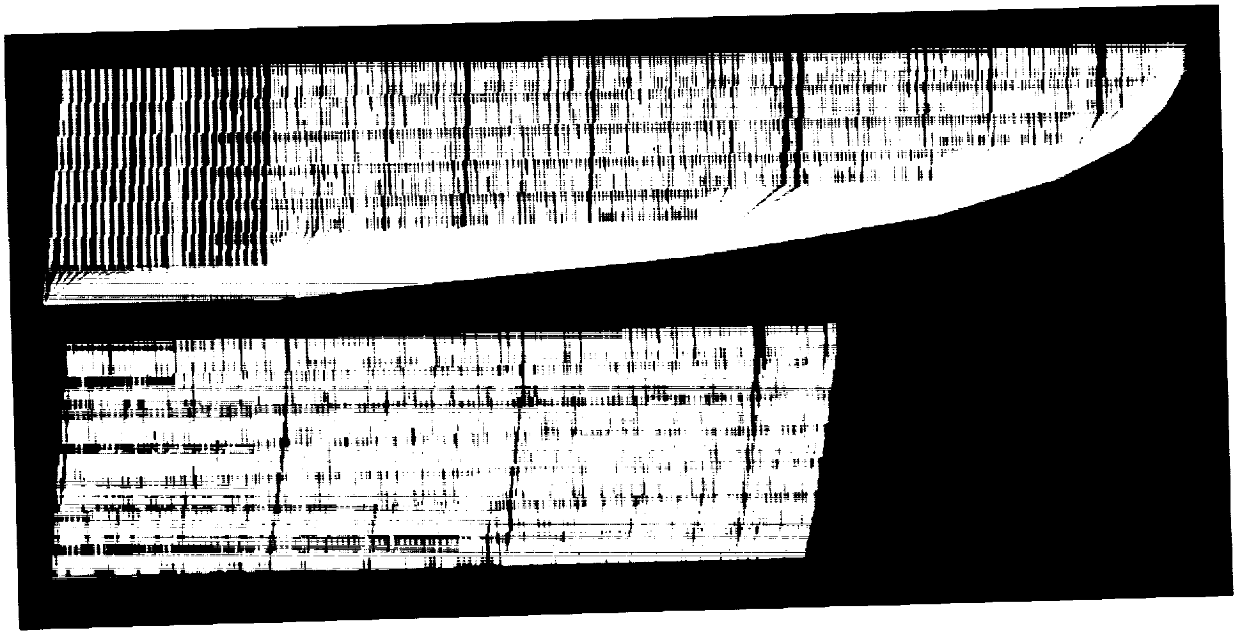
Unclas

G3/34 0048493

MCAT Institute
Final Report
95-15

Algorithms for parallel flow solvers on message passing architectures

Rob F. Van der Wijngaart



January 1995

NCC2-752

MCAT Institute
3933 Blue Gum Drive
San Jose, CA 95127

Algorithms for parallel flow solvers on message-passing architectures

Rob F. Van der Wijngaart

1 Introduction

The purpose of this project has been to identify and test suitable technologies for implementation of fluid flow solvers—possibly coupled with structures and heat equation solvers—on MIMD parallel computers. In the course of this investigation much attention has been paid to efficient domain decomposition strategies for ADI-type algorithms. In references [1, 2, 3] the near-optimal properties of the multi-partition strategy were explained, and efficient implementations were presented. These included solving the heat equation on rectilinear and curvilinear grids on the Intel iPSC/860, and solving the NAS scalar penta-diagonal parallel benchmark on the iPSC/860, Paragon, IBM SP2, and on a network of Silicon Graphics workstations connected through Ethernet. Multi-partitioning derives its efficiency from the assignment of several blocks of grid points to each processor in the parallel computer. A coarse-grain parallelism is obtained, and a near-perfect load balance results.

By contrast, in the uni-partitioning strategy every processor receives responsibility for exactly one block of grid points instead of several. This necessitates fine-grain pipelined program execution in order to obtain a reasonable load balance. Although fine-grain parallelism is less desirable on many systems, especially high-latency networks of workstations, uni-partition methods are still in wide use in production codes for flow problems. Consequently, it remains important to achieve good efficiency with this technique that has essentially been superseded by multi-partitioning for parallel ADI-type algorithms.

Another reason for the concentration on improving the performance of pipeline methods is their applicability in other types of flow solver kernels that have a stronger implied data dependence than ADI, such as SSOR [4], or flux-vector-split methods [5].

2 Results and conclusions

An important feature of fine-grain parallel applications on MIMD distributed-memory computers is the fact that there is no global hardware-supported synchronization. This means that communications may be initiated on certain processors, while the result is not expected (yet) by the receiving processors. The trap mechanism for unexpected messages incurs certain overheads which may lead to significant dynamic load imbalances when implementing pipeline algorithms. The effect of these overheads gets aggravated by fast feeding of the pipeline by the first processor, and substantial improvements can be obtained by artificially slowing down this first processor.

Analytical expressions can be derived for the size of the dynamic load imbalance incurred in traditional pipelines. From these it can be determined what is the optimal first-processor retardation that leads to the shortest total completion time for the pipeline process. Theoretical predictions of pipeline performance with and without optimization match experimental observations on the iPSC/860 very well.

Analysis of pipeline performance also highlights the effect of uncareful grid partitioning in flow solvers that employ pipeline algorithms. If grid blocks at boundaries are not at least as large in the wall-normal direction as those immediately adjacent to them, then the first processor in the pipeline will receive a computational load that is less than that of subsequent processors, magnifying the pipeline slowdown effect. Extra compensation is needed for grid boundary effects, even if all grid blocks are equally sized.

The results of the above investigations are described in references [6] and [7], which are attached to this report as appendices.

References

[1] R.F. Van der Wijngaart, *Efficient implementation of a 3-dimensional ADI method on the iPSC/860*, Proceedings Supercomputing '93, Portland, OR, November 93

[2] R.F. Van der Wijngaart, T. Phung, E. Barszcz, *Three implementations of the NAS scalar penta-diagonal benchmark*, submitted for presentation at Supercomputing '94, November 1994

[3] M.H. Smith, R.F. Van der Wijngaart, *Granularity and the parallel efficiency of flow solution on distributed computer systems*, 25th AIAA Fluid Dynamics Conference, Colorado Springs, CO, June 20-23, 1994

[4] S.E. Rogers, D. Kwak, *Steady and unsteady solutions of the incompressible Navier-Stokes equations*, AIAA Journal, Vol. 29, No. 4, 1991, pp. 603–610

[5] G.H. Klopfer, G.A. Molvik, *Conservative multizonal interface algorithm for the 3-D Navier-Stokes equations*, AIAA Paper 91-1601, AIAA 10th Computational Fluid Dynamics Conference, Honolulu, HI, June 1991

[6] R.F. Van der Wijngaart, S.R. Sarukkai, P. Mehra, *Analysis and Optimization of Software Pipeline Performance on MIMD Parallel Computers*, submitted for publication in Institute of Electrical and Electronics Engineers (IEEE) Journal of Parallel and Distributed Computing

[7] R.F. Van der Wijngaart, S.R. Sarukkai, P. Mehra, *The Effect of Interrupts on Software Pipeline Execution on Message-passing Architectures*, submitted for presentation at Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, July 19-21, 1995

Analysis and Optimization of Software Pipeline Performance on MIMD Parallel Computers

Rob F. Van der Wijngaart*, Sekhar R. Sarukkai†, Pankaj Mehra†
NASA Ames Research Center, Moffett Field, CA 94035

Abstract

Pipelining is a common strategy for extracting parallelism from a collection of independent computational tasks. Filling the pipeline creates an inevitable performance penalty. When implemented on MIMD parallel computers that transfer messages asynchronously, pipeline algorithms suffer an additional slowdown; processor interrupts cause a wave-like propagation of delays. This phenomenon, which has been observed experimentally using the AIMS performance monitoring system, is investigated analytically, and an optimal correction is derived to eliminate the wave. Increased efficiency through the correction is verified experimentally.

1 Introduction

Pipelining is a common strategy for extracting parallelism from a set of independent tasks, each of which is sequential in nature but is executed by multiple processors. A well-known example is the solution of large numbers of banded-matrix equations resulting from the discretization and approximate factorization of partial differential equations using the alternating-direction implicit (ADI) algorithm (e.g. [4]). Irrespective of the hardware platform on which a pipeline algorithm is implemented, a delay proportional to the number of processors that share the sequential data dependence is incurred. In the case of dedicated hardware pipelines such as those employed in traditional vector supercomputers, a result is produced during every clock cycle once the pipeline is full. We show that software pipelines implemented on MIMD (Multiple Instruction/Multiple Data) distributed-memory parallel computers with nonzero process interrupt times also suffer nonlinear delays that make them unattractive when the computational work per pipe segment is small. Such machines (e.g. IBM SP/2, Intel iPSC/860 and Paragon) constitute an important class of parallel computers.

*MCAT Institute
†Recom Technologies

Here we investigate the structure of those delays based on a simple parallel performance model. Building on the results of our investigation, an optimal strategy for reducing the delays is proposed and verified.

2 Algorithm and performance model

We consider a model problem consisting of a set of N identical independent tasks, $\{w^k \mid k = 1, N\}$, to be completed by p identical processors. Each task is divided uniformly into a set of p subtasks, $\{w_j^k \mid j = 1, p\}$. Every subtask w_j^k is assigned to processor j. A data dependence exists among the subtasks; subtask w_j^k cannot be started before subtask w_{j-1}^k has been finished.

The pipeline algorithm is constructed as follows. Processor 1 completes subtask w_1^1 and sends a message to processor 2 indicating that subtask w_2^1 can commence. Subsequently, processor 1 completes subtask w_1^2 while processor 2 completes subtask w_2^1. After both subtasks are completed, processor 1 sends another message to 2, and processor 2 signals processor 3. This pattern is repeated, and after $p-1$ subtasks have been completed by processor 1 all processors are active, provided that $N \geq p$.

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.

2. The computational work associated with subtask w_j^k requires a constant period of c time units.

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.

Assumptions 1 and 2 will be relaxed later. Previous pipeline analyses (e.g. [1]) assume that messages always arrive before the receive has been posted, so that processors never need to wait for data. Our observations (see Figure 1) reveal that this is not true in practice, so we drop this assumption. 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 [3]. This kind of communication delay automatically provides the type of pipeline optimization described in section 4, but at the cost of implicit synchronization between all pipeline segments.

Assumption 4 is what distinguishes our model most prominently from previous models, such as the one presented in [2], which uses s and r_h, but not r_i. Interrupts generate dynamic load imbalances because they consume cpu time during intermediate pipeline stages.

3 Performance analysis

We use the performance-monitoring package AIMS [5] (Automated Instrumentation and Monitoring System) to visualize 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 sections signify that a processor is performing

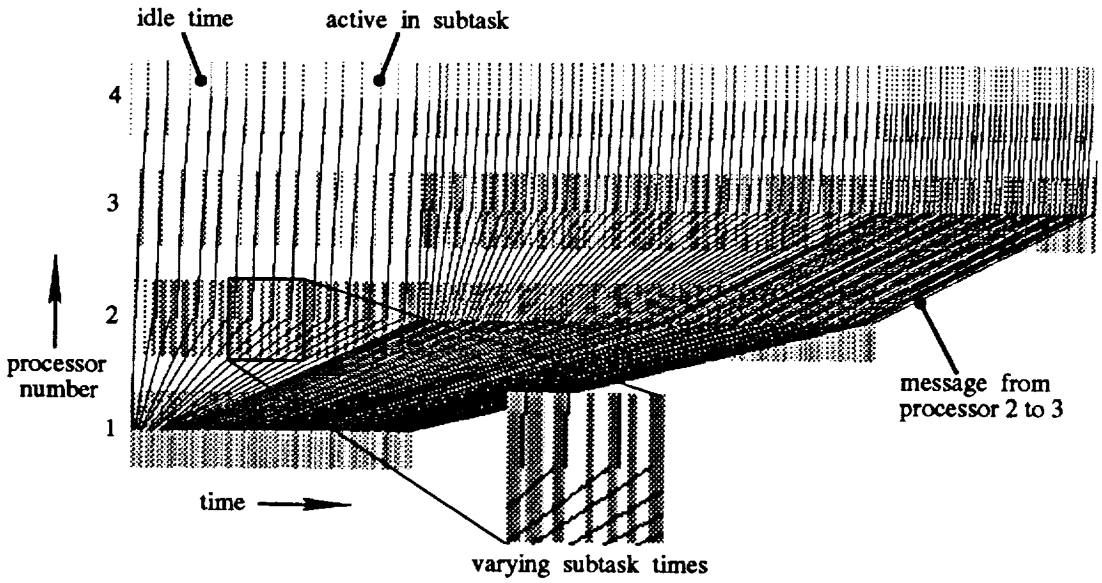


Figure 1: AIMS processor activity status

a subtask, whereas 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 originate on the AIMS time line 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 time spent within subtasks varies (see inset), as does the amount of time spent waiting in between subtasks. This variation is not due to variations in subtask execution times, but shows because AIMS does not explicitly monitor processor interrupts due to arriving messages.

A clear fan-out of message transfer lines is visible between processor 1 and processor 2, 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. In general, each processor experiences four pipeline phases, which are identified schematically in Figure 2. They are discussed below in reverse chronological order.

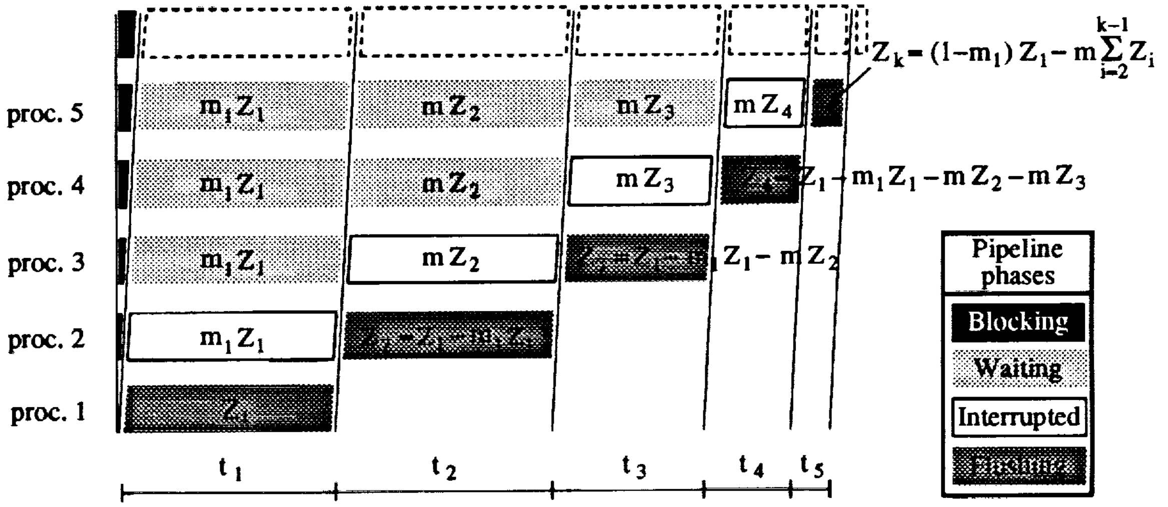


Figure 2: Four pipeline processor phases

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; 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, see below).

INTERRUPTED. The processor features long subtask 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.

WAITING. This phase is dominated by waiting for messages to arrive from the predecessor. Subtasks on the predecessor take 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.

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.

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. Subtasks can be grouped together 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. 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

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

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

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

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 $j \geq 2$.

Equation 2 implies that $r_i < c + s$. The meaning of violation of this inequality is that processor 2 will be constantly interrupted 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.

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

$$Z_k = Z_1 - m_1 Z_1 - m \sum_{j=2}^{k-1} Z_j \,, \quad \text{with} \quad Z_2 = (1 - m_1) Z_1 \,, \quad Z_1 = N \,. \tag{4}$$

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

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

The total duration T_k^b of the blocked mode on processor k is determined as follows. We assume that subtask w_k^1 is interrupted exactly once (by subtask w_{k-1}^1), 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 subtask w_2^1. 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:

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

Substituting equations 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 time:

$$\delta \mathrm{t}_k \overset{\mathrm{def}}{=} \frac{T_k}{N(c+s)} - 1 = (1+\alpha+\beta)\frac{k-1}{N} + \frac{\alpha+\beta}{1+\alpha-\beta}\left\{1+\alpha-\beta\left(\frac{\beta}{1+\alpha}\right)^{k-2}\right\}\,. \tag{11}$$

The first term on the right hand side of equation 11 signifies the expected scaled completion delay time 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 time is:

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

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.

If the number of processors is small, then the timing for the last processor may differ significantly from the asymptotic value, or even from equation 11. This situation occurs often with pipelines resulting from three-dimensional domain decompositions (e.g. [4]). 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.

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

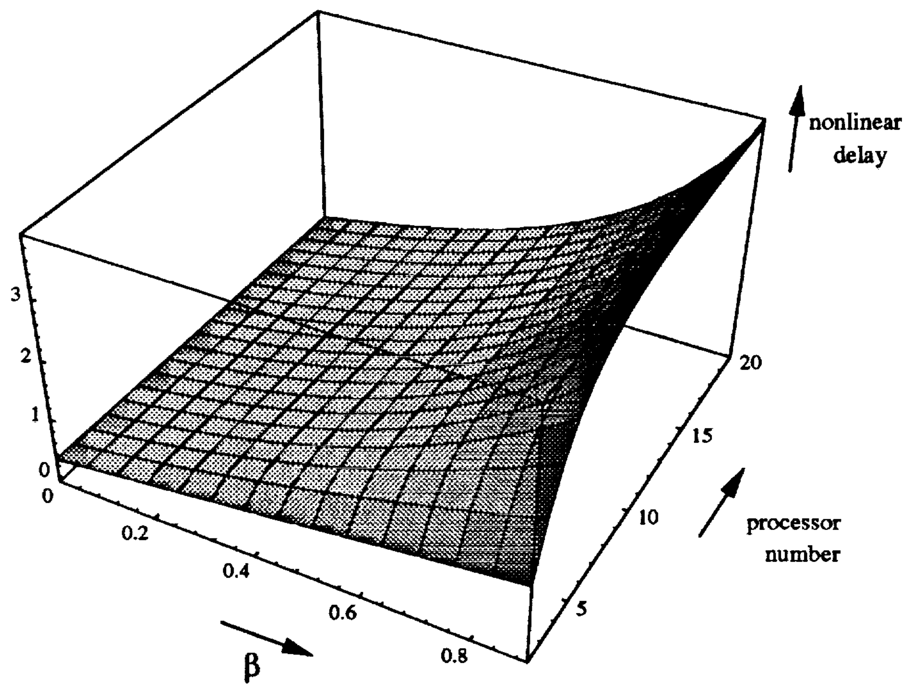


Figure 3: Scaled nonlinear delay for $\alpha = 0.3$

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

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

If $s < r_i$ there will still be a fan-out, and $t_p \neq 0$. Rewriting equation 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 equation 5), we obtain for the number of subtasks remaining during the flushing phase:

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

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

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

Introducing a third relative overhead:

$$\gamma = \frac{s}{c+s}, \tag{18}$$

we find the following scaled delay time:

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

Equations 13 and 19 can be combined to form:

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

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

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

and for the last processor in the pipeline:

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

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 time processor 2 is forced to wait for data from processor 1. Consequently,

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

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:

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

Introducing a scaled padding:

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

we obtain the following expression for the scaled completion delay:

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

Interestingly, for $k = 2$ (the second processor in the pipeline) we find that $\delta \mathrm{t}_k' = \delta \mathrm{t}_k$, 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. Equation 32 represents a linear function in τ, whose extremal values are attained at the boundaries of the interval defined by equation 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 time 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:

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

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:

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

4.2 Optimal grain size

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 equation 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:

$$\begin{aligned} N &\leftarrow N/n \\ c &\leftarrow nc \end{aligned}$$

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

This leads to the following final completion time:

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

The optimal subtask grouping size is obtained by setting

$$\frac{\partial T_p}{\partial n} = 0\,, \qquad (38)$$

which yields:

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

and the corresponding optimal padding per subtask is:

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

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 Verification

In order to verify the validity of our performance model we use a simple test program in which the amount of computational work per pipeline segment can be varied. The program is run on the Intel iPCS/860 and uses blocking send and receive calls of the NX message passing library to pass zero-length messages. The node numbering in the pipelines is such that only nearest-neighbor communication occurs, which eliminates contention. The number of pipeline tasks does not have an important qualitative influence on the performance; it is kept fixed ($N = 256$) in this experimental investigation.

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 is printed below.

```
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);
   }
}
```

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. The 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

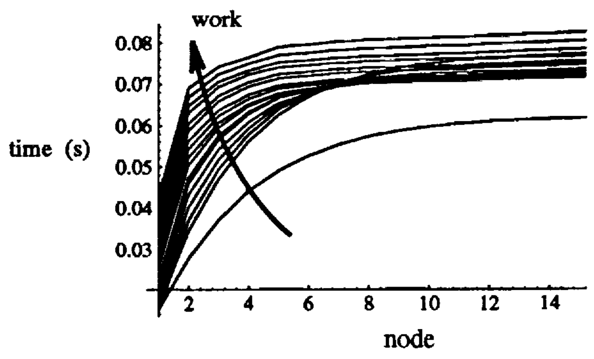
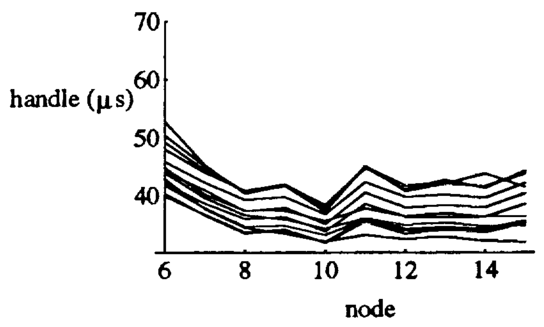


Figure 4: Completion times for increasing arithmetic loads

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 equation 9 we derive that $r_i + r_h = T_2/(N+1) - T_1/N$,



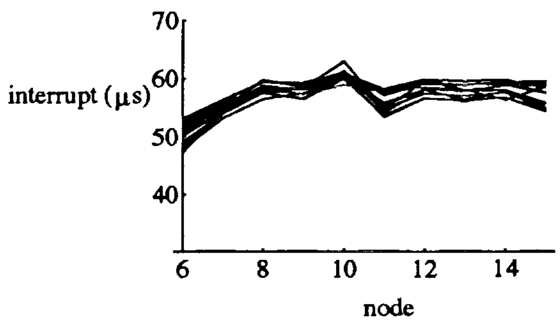
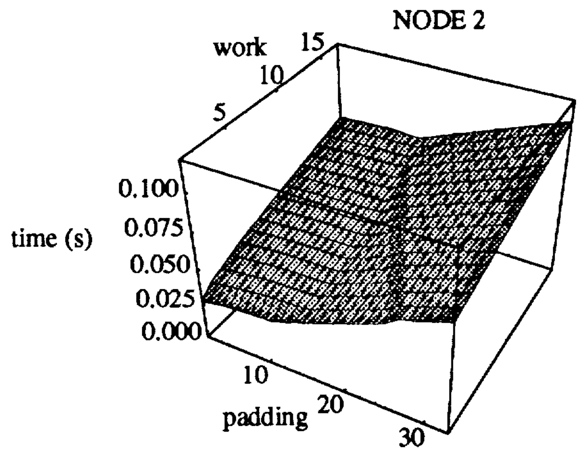


Figure 5: Measured receive overheads

which can be combined with equation 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 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 for node 15, we obtain the values of $r_h = 39\mu s$ and $r_i = 57\mu s$. 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 equation 32, the completion time on



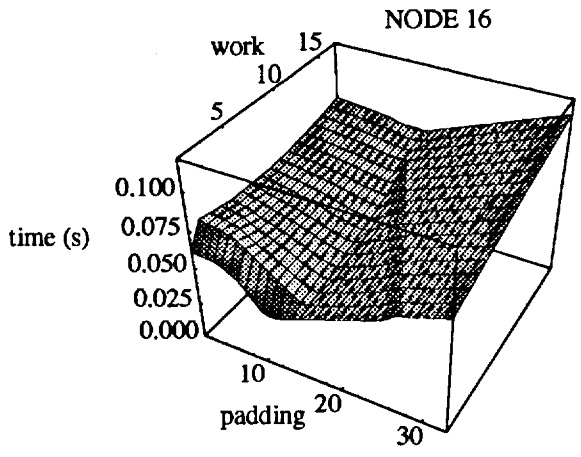


Figure 6: Completion times on the second and last pipeline nodes

node 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. In Figure

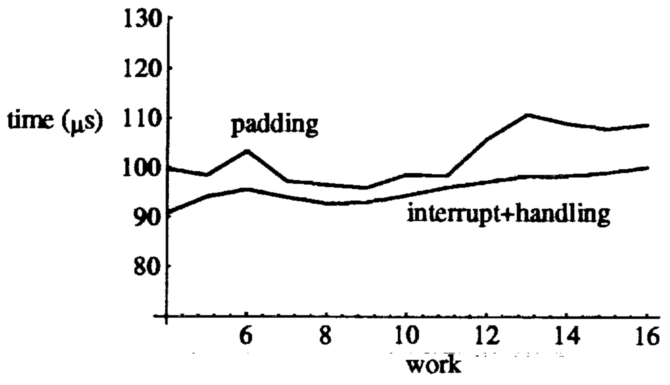


Figure 7: Comparison of receive overheads and optimal padding

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.

The achieved improvements in execution time for optimal padding versus no padding are plotted in Figure 8, alongside with 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 computation) in this figure is now given in microseconds instead of in number of times the loop body in the test program is executed. Agreement between predicted and measured speed-ups is quite good.

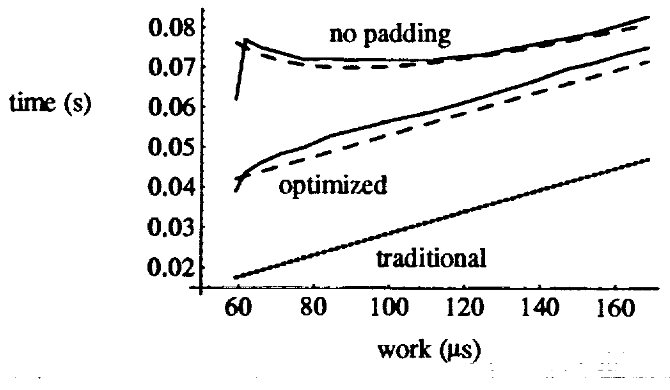


Figure 8: Predicted (dashed) and measured (solid) completion times with and without padding, and traditional model (dotted)

In the same figure we also depict expected pipeline performance using traditional analysis (dotted line), which takes latency into account, but ignores the effect of processor interrupts. The traditional model, which assumes that all message overheads are borne by the communication network, significantly underpredicts completion times, even when compared to the case of optimal padding.

6 Conclusions

In this paper we have investigated the performance of software pipelines on MIMD parallel computers with message passing based on interrupts. Based on observations obtained using the performance monitoring and visualization tool AIMS, an analytical model for the pipeline behavior was formulated. A salient feature of the model is that it considers handling and interruption overheads associated with receive operations. Four phases of pipeline operation are identified, each with different communication characteristics. Recurrences identified in some phases give rise to nonlinear delays. Each phase is analyzed independently, and predicted total completion time is obtained by summing the individual contributions. Comparison of predicted execution with measured execution is favorable.

In addition to providing accurate predictions, the model also points to possible optimization of the implementation of parallel software pipelines. The optimization involves reducing the frequency with which the first processor in the pipeline sends messages to its successor. This artificial slow-down of the first processor actually improves overall program performance. The predicted magnitude of optimal first-processor retardation and the resulting total speed-up are again verified experimentally.

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.

References

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

[2] Adve, V. et al. Integrated performance analysis of data-parallel programs, *Workshop on Debugging and Performance Tuning for Parallel Computing Systems*, Sponsor: Los Alamos National Laboratory, Chatham, MA, 1994

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

[4] Van der Wijngaart, R.F. Efficient implementation of a 3-dimensional ADI method on the iPSC/860, *Proc. Supercomputing '93*, Portland, OR, 1993, pp. 102–111

[5] Yan, J.C. Performance tuning with AIMS—An automated instrumentation and monitoring system for multicomputers, *Proc.* 27th *Hawaii International Conference on System Sciences* **II**, Wailea, HI, 1994, pp. 625–633,

The Effect of Interrupts on Software Pipeline Execution on Message-passing Architectures

Rob F. Van der Wijngaart*, Sekhar R. Sarukkai†, Pankaj Mehra†
NASA Ames Research Center, Moffett Field, CA 94035

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 is presented that fully explains these load imbalances, and that allows for their removal. The results of applying this optimization to a tri-diagonal equation solver on the Intel iPSC/860 and Paragon are presented.

1 Introduction

It is well-known that software pipeline performance concerns a trade-off between granularity and exploited parallelism; pipeline fill time may be decreased at the expense of a larger number of communications. Indeed, experiments show that an optimum is usually achieved at less than the maximum exploitable parallelism. Here we demonstrate that part of the delay of fine-grain software pipelines implemented on MIMD distributed-memory parallel computers with nonzero process interrupt times (e.g. IBM SP/2, Intel iPSC/860) is attributable to dynamic load imbalances created by interrupts. We investigate the structure of those delays based on a simple parallel performance model. Building on the results of our investigation, an optimal strategy for reducing the delays is proposed and verified.

The same strategy is subsequently employed to optimize the solver phase of an important class of algorithms for the solution of partial differential equations whose parallel implementation necessitates fine-grain pipelines. An example is the Alternating Direction Implicit method in the widely used parallel flow solver POVERFLOW [6]. Since processor speeds are generally increasing faster than network bandwidths, it is envisioned that fine-grain pipelines will keep gaining prominence as medium-grain pipeline applications move toward the fine end of the granularity spectrum.

In order to describe previous and the current work on software pipelines, we consider a model problem consisting of N identical independent tasks, $\{w^k \mid k = 1, N\}$, to be completed by p identical processors. Each task is divided uniformly into p subtasks, $\{w_j^k \mid j = 1, p\}$. Every subtask w_j^k is assigned to processor j. A data dependence exists among the subtasks; subtask w_j^k cannot start before subtask w_{j-1}^k has finished. The pipeline algorithm is constructed as follows. Processor 1 completes subtask w_1^1 and sends a message to processor 2 indicating that subtask w_2^1 can commence. Subsequently, processor 1 completes subtask w_1^2 while processor

*MCAT Institute, †Recom Technologies

2 completes subtask w_2^1. After both subtasks are completed, processor 1 sends another message to 2, and processor 2 signals processor 3. This pattern is repeated, and after $p-1$ subtasks have been completed by processor 1, all processors are active, provided that $N \geq p$.

An intuitively appealing description of this pipeline process assumes that each subtask requires a fixed, constant amount of cpu 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 results in a perfectly synchronized pipeline operation in which no processor ever waits for data once the first message is received; pipeline fill times and total completion times are linear functions of the processor number p.

We use the performance-monitoring tool AIMS [5] (Automated Instrumentation and Monitoring System) to visualize 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 sections signify that a

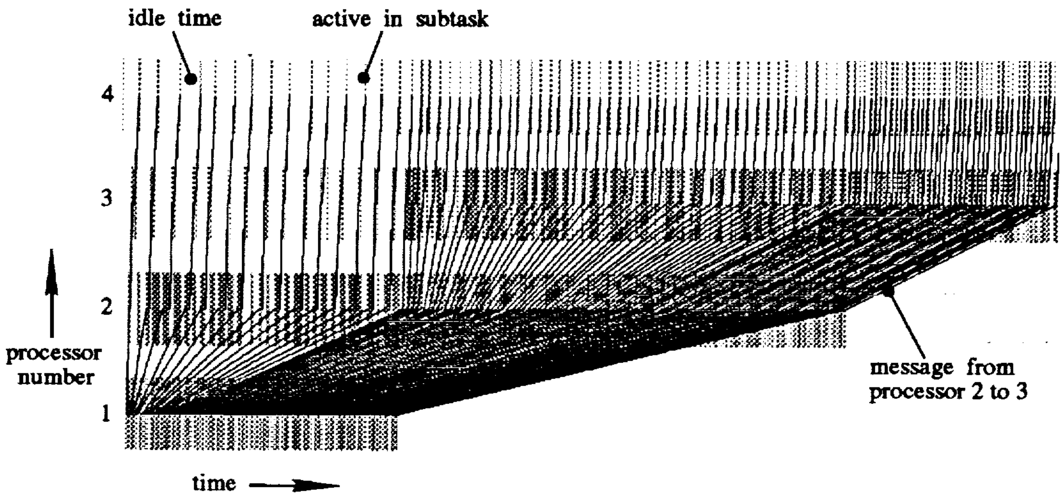


Figure 1: AIMS processor activity status for pipeline algorithm

processor is performing a subtask, whereas 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.

Obviously, total completion time is not linear in p, and not all subtasks last equally long, since not all message transfer lines are parallel. Moreover, in contrast with previous studies (e.g. King et al. [1]) 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 refinement of the simple model is needed.

A more realistic model such as that by Adve et al. [2] 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.

2 Performance analysis

Here we postulate a communication model that completely explains the observed pipeline behavior, and that offers a means for optimization described in setion 3. Its most important feature is the occurrence of interrupt events that generate dynamic load imbalances.

Assumptions:
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. It is not essential for the analysis, but leads to somewhat simpler algebra.
The computational work associated with subtask w_j^k requires a constant period of c time units.
When a message is sent by a processor a constant non-overlappable send overhead of s time units is incurred immediately by that processor.
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.
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.

In Figure 1 we observe a clear fan-out of message transfer lines between processor 1 and processor 2, 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. In general, each processor experiences four pipeline phases, which are identified schematically in Figure 2. They are discussed below, using the new communication model.

FLUSHING. The processor is not interrupted, either because there is no predecessor (processor 1), or because the predecessor has already completed all its subtasks. Processors in this phase still need to handle already arrived messages (except processor 1), perform the remaining subtasks, and send messages to their successor; since the preceding processor has fired messages from the same no-interrupt state, subtasks take equally long on successive processors in the pipeline (except on processor 2, see below).

INTERRUPTED. Long subtask duration is caused by high-frequency interruption by the predecessor. The predecessor is in the flush phase, and sends messages rapidly. Processor 1 does not exhibit the interrupt phase.

WAITING. This phase is dominated by waiting for messages to arrive from the predecessor. Subtasks on the predecessor take a long time to complete, either because that processor is in the wait phase itself, or because it is being interrupted frequently; subtasks take equally long on successive processors in the pipeline. Processors 1 and 2 do not exhibit this phase.

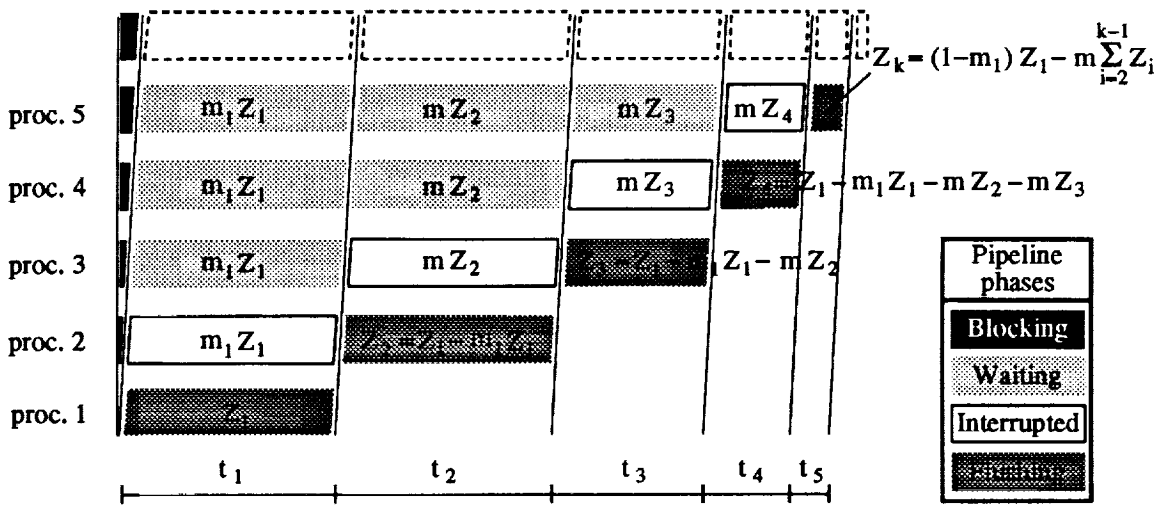


Figure 2: Four pipeline processor phases

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.

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, *blocked* and *active*. 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. Subtasks can be grouped together 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. 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 $c + s + r_h + r_i/m_1 = (c+s)/m_1$, so

$$m_1 = \frac{c + s - r_i}{c + s + r_h}. \tag{1}$$

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. 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

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

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

$$Z_k = Z_1 - m_1 Z_1 - m \sum_{j=2}^{k-1} Z_j\,, \quad \text{with} \quad Z_2 = (1 - m_1) Z_1\,, \quad Z_1 = N\,. \tag{3}$$

This simple recursion is most easily solved by computing $Z_{k+1} - Z_k$. It follows that $Z_{k+1} = (1 - m)Z_k$, so $Z_k = N(1 - m_1)(1 - m)^{k-2}$, $k \geq 2$. The total duration T_k^a of the active mode on processor k is now computed as

$$\begin{aligned} T_k^a &= t_1 + \sum_{j=2}^{k-1} t_j + t_k \\ &= (c + s)Z_1 + (c + s + r_h + r_i/m)m \sum_{j=2}^{k-1} Z_j + (c + s + r_h)Z_k \\ &= N(c + s) + N(1 - m_1)(c + s + r_h + \left(1 - (1 - m)^{k-2}\right) r_i/m)\,, \;\; k \geq 2\,. \end{aligned} \tag{4}$$

The total duration T_k^b of the blocked mode on processor k is easily determined:

$$T_k^b = (k - 1)(c + s + r_i + r_h)\,. \tag{5}$$

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

$$\begin{aligned} T_k &= (k - 1)(c + s + r_h + r_i) + N(c + s) + \\ &\quad N(1 - m_1)(c + s + r_h + \left(1 - (1 - m)^{k-2}\right) r_i/m)\,, \;\; k \geq 2\,. \end{aligned} \tag{6}$$

Substituting equations 2 and 1 into 6 and defining the two relative interrupt and handling overheads $\alpha = r_h/(c + s)$ and $\beta = r_i/(c + s)$, we obtain the scaled delay time:

$$\delta t_k \stackrel{\text{def}}{=} \frac{T_k}{N(c + s)} - 1 = (1 + \alpha + \beta)\frac{k - 1}{N} + \frac{\alpha + \beta}{1 + \alpha - \beta}\left\{1 + \alpha - \beta\left(\frac{\beta}{1 + \alpha}\right)^{k-2}\right\}. \tag{7}$$

The first term on the right hand side of equation 7 represents the expected pipeline fill. The second term is the additional nonlinear delay due to the wave-like propagation of the effects of message interrupts (see Figure 2).

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

$$\delta t_k = (1+\alpha+\beta)\frac{k-1}{N} + \frac{(\alpha+\beta)(1+\alpha)}{1+\alpha-\beta}\,. \tag{8}$$

3 Optimization

It follows from the analysis in section 2 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 spread its messages evenly during the second processor's execution. This suggests that the pipeline algorithm can be optimized by artificially increasing the amount of work performed by the first processor for each subtask.

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. The symbol $'$ indicates perturbations due to padding. No benefits can be obtained by increasing the padding beyond the sum of handling and interrupt overheads, since then processor 2 is forced to wait for data from processor 1. Consequently,

$$0 \le c' \le r_i + r_h\,. \tag{9}$$

If we assume that padding operations within each subtask on processor 1 take place *after* each send operation, then processor block times remain unchanged, *i.e.*, $(T_k^b)' = T_k^b$. In addition, we find that $t_1' = N(c'+c+s)$. Expressions for the numbers of subtasks per period stay the same, but the definition of m_1 changes:

$$m_1' = \frac{c'+c+s-r_i}{c+s+r_h}\,. \tag{10}$$

The total time spent on node k becomes:

$$\begin{aligned} T_k' &= (k-1+N)(c+s)+(k-1)(r_i+r_h)+Nc'+ \\ &\quad N(1-1/m_1)(c+s+r_h+\left(1-(1-1/m)^{k-2}\right)m\ r_i) \quad ,k\ge 2\,. \end{aligned} \tag{11}$$

Introducing a scaled padding: $\tau = c'/(c+s)$, we obtain the following expression for the scaled completion delay:

$$\begin{aligned} \delta t_k' &= (1+\alpha+\beta)\frac{k-1}{N}+\tau+\frac{\alpha+\beta-\tau}{1+\alpha-\beta}\left\{1+\alpha-\beta\left(\frac{\beta}{1+\alpha}\right)^{k-2}\right\} \\ &= \delta t_k + \tau\frac{\beta}{1+\alpha-\beta}\left\{\left(\frac{\beta}{1+\alpha}\right)^{k-2}-1\right\}\,. \end{aligned} \tag{12}$$

Equation 12 represents a linear function in τ, whose extremal values are attained at the boundaries of the interval defined by equation 9. We can assume that $\beta < 1$ (*i.e.* $r_i < c+s$), so the coefficient of τ is negative. Consequently, the total delay time is minimized by maximizing τ, yielding:

$$\tau = \alpha+\beta \quad \text{or} \quad c' = r_i + r_h\,. \tag{13}$$

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 $m_1' = 1$, 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:

$$\delta t_p^{opt} = (1+\alpha+\beta)\frac{p-1}{N} + \alpha + \beta' \tag{14}$$

which has the appearance of a synchronized pipeline result with a slightly modified subtask duration.

4 Verification and numerical application

In order to verify the validity of our performance model we use a simple test program in which the amount of computational work per pipeline segment can be varied. The program is run on the Intel iPCS/860 and uses blocking send and receive calls of the NX message passing library to pass zero-length messages. The node numbering is such that only nearest-neighbor communication occurs, which eliminates contention. The number of pipeline tasks does not have an important qualitative influence on the performance; it is kept fixed ($N = 256$) in this experimental investigation.

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 is printed below.

```
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);}
}
```

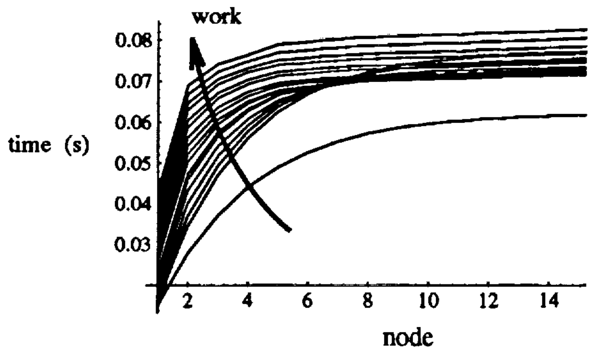


Figure 3: Completion times for increasing arithmetic loads

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 3 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 3 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 equation 6 we derive that $r_i + r_h = T_2/(N+1) - T_1/N$, which can be combined with equation 8 for larger values of k and $k+1$ to compute α and β (and hence r_i and r_h) separately. By averaging the results of these computations for node 15, we obtain the values of $r_h = 39\mu s$ and $r_i = 57\mu s$. Qualitative validity of the performance model derived above is demonstrated in Figure 4, which shows completion times on nodes 2 and 16, respectively, for the entire (**work,padding**) parameter space. As predicted by equation 12, the completion time on node 2 stays

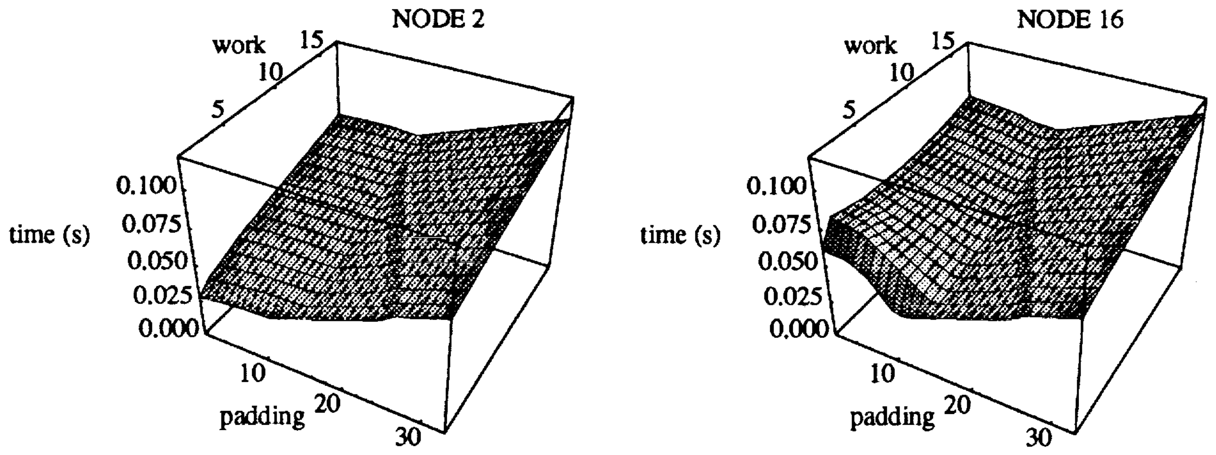


Figure 4: Completion times on the second and last pipeline nodes

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.

The achieved improvements in execution time for optimal padding versus no padding are plotted in Figure 5, alongside with 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. Agreement between predicted and measured speed-ups is quite good. In the same figure we also depict expected performance using traditional analysis (dotted line), which takes latency into account but ignores the effect of interrupts. The traditional model, which assumes that all message overheads are borne by the communication network, significantly underpredicts completion times, even when compared

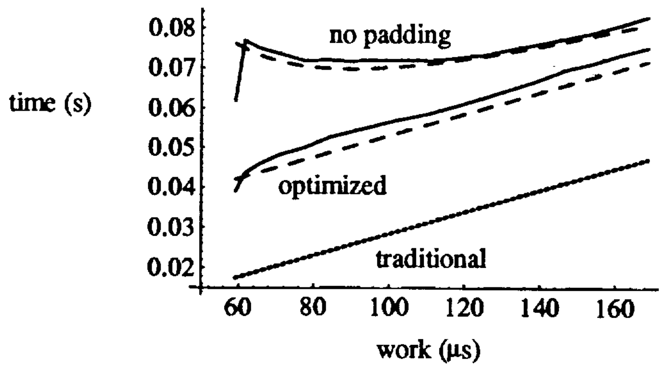


Figure 5: Predicted (dashed) and measured (solid) completion times with and without padding, and traditional model (dotted)

to the case of optimal padding.

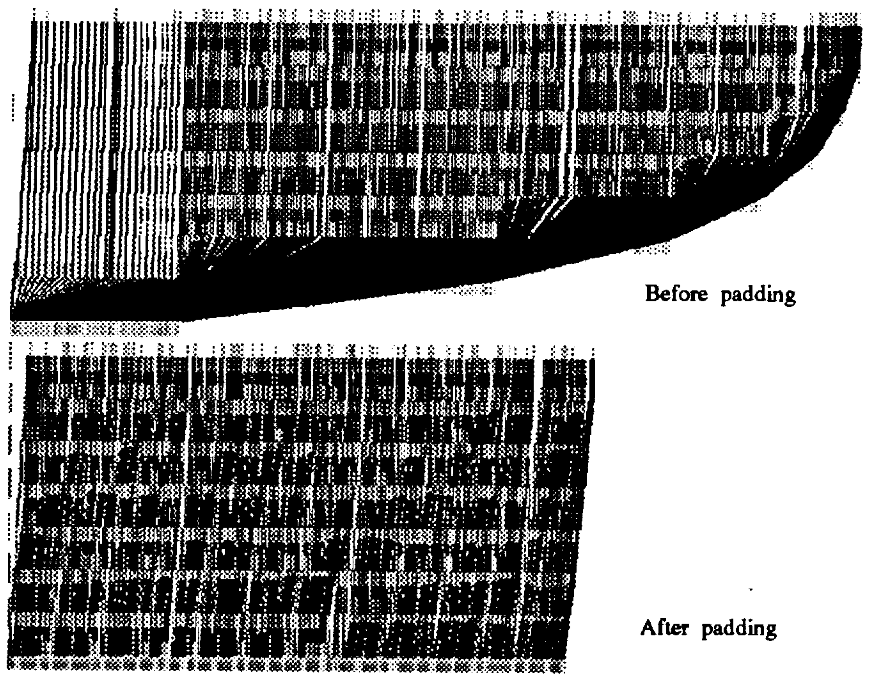


Figure 6: Optimization of tri-diagonal solver on Intel Paragon

We now apply padding to a problem derived from the solution of 3-dimensional partial differential equations (e.g. [4]). It concerns a set of 256 independent tri-diagonal matrix equations, each of rank 128. The problem is solved on 8 nodes of an Intel Paragon XP/S without message co-processor. Pipelining is obtained by having every node solve $1/8^{th}$ of each matrix equation using Gaussian elimination without pivoting. We only consider the forward elimination, which requires 6 multiply/adds and 1 division per matrix row. Each node passes only two 8-byte numbers for each matrix equation to its successor. Figure 6 shows the AIMS performance profile before and after padding. Although the padding is slightly more than optimal (execution time on node 2 has increased), the performance gain is already more than 30%. Improvements will be more dramatic for the even finer-grained backsubstitution phase (only 2 multiply/adds and no divisions per matrix row).

These exercises also indicate potential for improvement of pipeline algorithms in application programs such as POVERFLOW [6], which solves a collection of penta-diagonal matrix equations derived from the discretization of the compressible Navier-Stokes equations. Here 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, POVERFLOW 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.

5 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 was formulated. Salient features of the model are the handling and interrupt overheads associated with receive operations. Four phases of pipeline operation are identified, each with different communication characteristics. Recurrences identified in some phases give rise to nonlinear delays. Each phase is analyzed independently, and predicted total completion time is obtained by summing the individual contributions. Comparison of predicted and measured execution is favorable.

The model also points to possible optimization of the implementation of parallel software pipelines 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 total speed-up are again verified experimentally.

References

[1] C.-T. King, W.-H. Chou, L.M. Ni, *Pipelined data-parallel algorithms: Part I — Concept and modeling*, IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 4, October 1990

[2] V. Adve et al., *Integrated performance analysis of data-parallel programs*, Workshop on Debugging and Performance Tuning for Parallel Computing Systems, Chatham, MA, October 3-5, 1994

[3] W.C. Saphir, *Message buffering and its effect on the communication performance of parallel computers*, NASA Report RNS-94-004, NASA Ames Research Center, Moffett Field, CA, April 1994

[4] R.F. Van der Wijngaart, *Efficient implementation of a 3-dimensional ADI method on the iPSC/860*, Proceedings of Supercomputing '93, pp. 102-111, Portland, OR, November 15-19, 1993

[5] J.C. Yan, *Performance tuning with AIMS—An automated instrumentation and monitoring system for multicomputers*, Proceedings of the 27th Hawaii International Conference on System Sciences, Vol. II, pp. 625-633, Wailea, HI, January 4-7, 1994.

[6] J.S. Ryan, S.K. Weeratunga, *Parallel computation of 3-D Navier-Stokes flowfields for supersonic vehicles*, AIAA 93-0064, 31st Aerospace Sciences Meeting and Exhibit, Reno, NV, January 11-14, 1993