arrays. In this article, the following two-dimensional, second-order linear partial differential equation (PDE)

$$a(x,y)\frac{\partial^2 u}{\partial x^2} + b(x,y)\frac{\partial u}{\partial x} + c(x,y)\frac{\partial^2 u}{\partial y^2} + d(x,y)\frac{\partial u}{\partial y} + e(x,y)\frac{\partial^2 u}{\partial x \partial y} + f(x,y)u = g(x,y) \quad (1)$$

and its numerical solution via Successive Over-Relaxation (SOR) methods is considered. Given an initial estimate $u^{(0)}$, the SOR methods (7) obtain a refined estimate $u^{(R)}$ of the solution of Eq. (1) discretized over an $M \times N$ grid by using R iterations which iteratively improve each of the discretized solution estimate components $u_{m,n}^{(r)}$ by combining the previous estimate $u_{m,n}^{(r-1)}$ with recent estimates of its northern, western, eastern, and southern neighbors. Thus

$$u_{m,n}^{(r)} = u_{m,n}^{(r-1)} - \omega^{(r)}[\beta_{m,n,N} u_{m-1,n}^{(*N)} + \beta_{m,n,W} u_{m,n-1}^{(*W)} + u_{m,n}^{(r-1)} + \beta_{m,n,E} u_{m,n+1}^{(*E)} + \beta_{m,n,S} u_{m+1,n}^{(*S)} - \gamma_{m,n}] \quad (2)$$

for $r = 1, 2, \ldots, R$ and for all $(m, n) \in \Omega^\circ$, given the relaxation sequence $\omega^{(r)}$ for $r = 1, 2, \ldots, R$, an initial discretized solution estimate $u_{m,n}^{(0)}$ for all $(m, n) \in \Omega^\circ$ and boundary conditions $u_{m,n}^{(R)} = u_{m,n}^{(0)}$ for all $(m, n) \in \partial\Omega$ where

$$\Omega = \left\{ (m,n) \middle| \begin{matrix} m \in \{0, 1, \ldots, M+1\} \\ n \in \{0, 1, \ldots, N+1\} \end{matrix} \right\}$$

where Ω° and $\partial\Omega$ denote the interior and boundary of Ω, respectively, and where each sweeping ordering parameter $*_N$, $*_W$, $*_E$, and $*_S$ takes a value of r or $(r - 1)$ and implies a sequence of precedence among the computations of $u_{m,n}^{(r)}$. A family of parallel SOR algorithms is obtained by segmenting the SOR algorithms into arithmetic grains, parameterizing the assignment of the arithmetic grains to at most P parallel processes intended for execution on P processors, and parameterizing the number of arithmetic grains computed between communications events. To evaluate the complexity and performance of the parallel algorithms presented here, it is assumed that $\omega^{(r)}$ and R are known and that the discretization grid is static.

Because the numerical performance and parallelism of a given algorithm depend on the ordering parameters (8–11), the Jacobi (J), red–black Gauss–Seidel (RB), and natural Gauss–Seidel (GS) orderings are considered. In the Jacobi ordering, $*_N = *_W = *_E = *_S = r - 1$. Thus with the J ordering, all components at iteration r may be computed in parallel. In

ELLIPTIC EQUATIONS, PARALLEL OVER SUCCESSIVE RELAXATION ALGORITHM

Numerous numerical parallel techniques exist for solving elliptic partial differential equation discretizations (1–4). The most popular among these are parallel Successive Over-Relaxation (SOR) (5) and parallel multigrid methods (6) for a variety of parallel architectures, including shared memory machines, vector processors, and one- and two-dimensional

Table 1. Ordering Parameters

Sweeping Order		$*N$	$*W$	$*E$	$*S$
Jacobi (J)		$r-1$	$r-1$	$r-1$	$r-1$
Red–black (RB)	red	$r-1$	$r-1$	$r-1$	$r-1$
	black	r	r	r	r
Gauss–Seidel (GS)		r	r	$r-1$	$r-1$

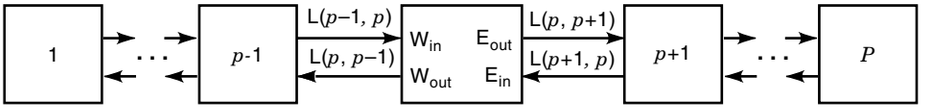

Figure 1. Linear array with bidirectional communication links.

the RB ordering, the components of u are divided into two groups; $u_{m,n}$ is red if $(m + n)$ is even, and black if $(m + n)$ is odd. Red components at iteration r are updated using black components from iteration $(r - 1)$, that is, $*_N = *_W = *_E = *_S = r - 1$. Black components at iteration r are updated using red components from iteration r, that is, $*_N = *_W = *_E = *_S = r$. Thus with the RB ordering, all red components may be computed in parallel followed by the computation of all black components in parallel. In the GS ordering, $*_N = *_W = r$, and $*_E = *_S = r - 1$, and thus all components with identical values of $(m + n)$ may be computed in parallel. These orderings are summarized in Table 1.

If the number of iterations R, which guarantee a solution of desired accuracy is not known, then a dynamic stop rule can be implemented by redefining an arithmetic grain to include accumulating the magnitudes of the terms in the parenthesis of Eq. (2) for each iteration and comparing the accumulation to a threshold.

The number of iterations R which guarantee a solution of desired accuracy depend on the relaxation sequence $\omega^{(r)}$. There are many relaxation schemes including static (7), unadaptive dynamic (7,12), global adaptive dynamic (13), and local adaptive dynamic (14–16). In the static and unadaptive dynamic cases, the relaxation sequence $\omega^{(r)}$ is known before execution of the SOR and therefore the evaluation of the SOR requires no computations other than those in Eq. (2). In the global adaptive dynamic and local adaptive dynamic cases, the relaxation sequence is computed as the SOR iterations proceed. In these cases, again an arithmetic grain can be redefined to incorporate the computations of such adaptive strategies.

The use of an adaptive grid is another strategy that can enhance SOR algorithm performance (17). This strategy computes an initial, crude, approximate solution on a coarse mesh with a low-order numerical method that is enriched until a prescribed accuracy is attained. Enrichment indicators, which are frequently estimates of the local discretization error, are used to control the adaptive process. Resources are introduced in regions having large enrichment indicators and are deleted from regions where indicators are low. This strategy can also be incorporated by redefining an arithmetic grain to include the calculation and usage of enrichment indicators.

To evaluate the complexity and performance of the parallel algorithms presented in this article, it is assumed that the relaxation sequence is known, that R is fixed and known, and that the discretization grid is static.

ARCHITECTURE AND ARCHITECTURAL PARAMETERS

The target architecture and associated software protocol consists of P processors connected in a linear array with bidirectional communication links, as shown in Fig. 1. The linear array was chosen for several reasons. First, it is among the least complex of all parallel architectures. If a parallel algorithm can be devised to execute efficiently on a linear array, then it is not necessary to consider more complicated architectures. Second, an algorithm developed for a linear-array topology is portable among architectures because it can be executed on topologies which include the linear array. Third, linear arrays require less hardware, are physically smaller, consume less power, require less cabling and backplane wiring, and are less expensive than more heavily connected topologies.

There are two communication links between processor p and processor $(p - 1)$ designated W_{in} (West in) and W_{out} (West out) on processor p. Likewise, there are two communication links between processor p and processor $(p + 1)$ designated E_{in} (East in) and E_{out} (East out) on processor p. The unidirectional link from processor p to processor q is designated $L(p, q)$. Each processor executes an instruction stream consisting of arithmetic and message initiation instructions. Input and output message initiations must be paired for two processors to communicate and exchange data. Communication between processors is synchronized. When data is passed between two processors, the output processor is blocked until the input processor is ready and vice versa (18). Furthermore, output messages are not initiated until the last word of a message has been computed.

Total latency is a combination of arithmetic latency and communication latency. Each processor requires time τ_a to execute an arithmetic instruction where τ_a includes the cost

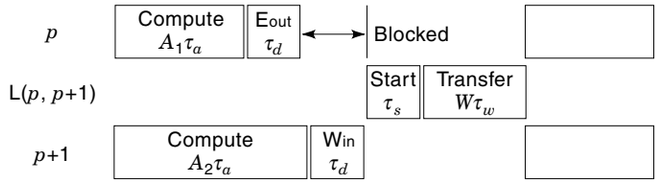

Figure 2. Nonconcurrent message startup blockage from p to $(p + 1)$.

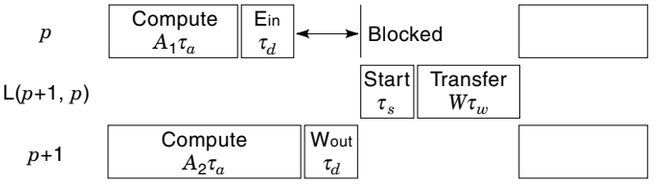

Figure 3. Nonconcurrent message startup blockage from $(p + 1)$ to p.

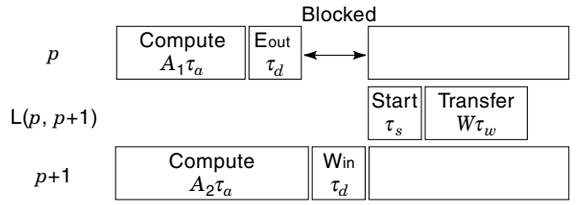

Figure 4. Concurrent message startup blockage.

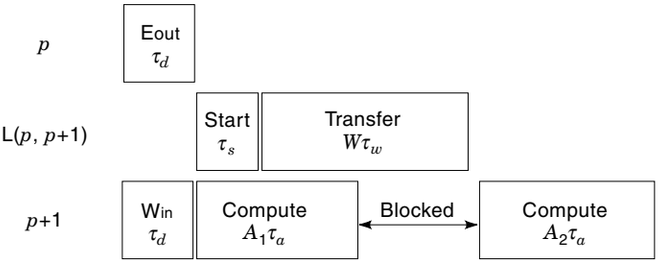

Figure 6. Data-dependency blockage.

of instruction fetch and decode, operand fetch and save, caching, operand index calculation, loop overhead, etc.. Each processor requires overhead time τ_d to initiate a message, where τ_d includes the cost of initializing source address, destination address, and message length registers, possible buffer allocation, etc. A communication link requires time $\tau_c(W) = \tau_s + W\tau_\omega$ to transfer a W-word message across a link where τ_s is the message start-up time and τ_ω is the per word transfer time if the other processor participating in the communication is ready for the message transfer. If the other processor is not ready, then the link blocks and transfer of the message is delayed. Message startup time τ_s includes the time to synchronize clocks, transfer header information, etc.

The capabilities of the P processors classify the architecture as either *nonconcurrent* or *concurrent.* The presence or absence of concurrency is usually determined by the presence or absence of a direct memory access (DMA) unit.

Nonconcurrent Architecture

In nonconcurrent architecture, the P processors perform either the execution of arithmetic instructions, message initiation instructions, or unidirectional communications across one communication link at any given time. When a (synchronized) communication takes place from processor p to processor $(p + 1)$ across a communication link, the processor which finishes its corresponding message initiation first, say p, blocks as seen in Fig. 2. When the other processor $(p + 1)$ finishes its message initiation, the link unblocks and message startup occurs on the communication link for a duration τ_s. At the conclusion of startup, words are transferred across the communication link with a latency of τ_ω for each word until the message transfer is complete. Arithmetic and message-initiation processing remains blocked throughout the message transfer. At the conclusion of the message transfer, arithmetic or message-initiation processing resumes on both processors (dashed boxes). Figure 3 shows a similar situation with the direction of communication reversed, that is, data is transferred from $(p + 1)$ to p. In this case, it is still the processor that finishes message initiation first (p) which blocks.

Concurrent Architecture

In concurrent architecture, the processors are capable of executing arithmetic instructions or message-initiation instructions simultaneously with bidirectional communications on all communication links. A processor that finishes message initiation first blocks, as in the nonconcurrent case. However, after the second processor finishes message initiation, execution of arithmetic instructions or message initiation may resume, as shown in Fig. 4 in addition to the unblocking of the communication link.

Initiation of a message on a communication link is blocked until any message in progress on that link completes. For example, message initiations from processor p to processor $(p + 1)$ are blocked on both p and $(p + 1)$ until the transfer from p to $(p + 1)$ is complete, as shown in Fig. 5.

If arithmetic instructions depend on message data, processing is blocked until message completion. For example in Fig. 6, processor $(p + 1)$ executes A_1 arithmetic instructions, blocks until the message transfer is complete, and then executes A_2 arithmetic instructions which are assumed to depend on message data.

Note that if instructions are properly coordinated among processors, then it is possible for a processor to execute arithmetic instructions simultaneously with the transfer of messages on all communications links, as shown in Fig. 7.

THE PARAMETERIZED FAMILY OF SOR ALGORITHMS

An *arithmetic grain,* denoted by its output $u^{(r)}_{m,n}$, consists of the operations of Eq. (2) for fixed m, n, and r. The arithmetic complexity and communication among grains are summarized in Table 2. Thus R iterations of SOR consist of MNR arithmetic grains whose execution require $11MNR$ operations.

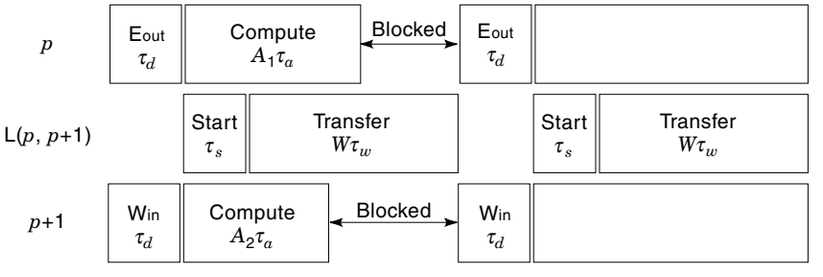

Figure 5. Message-initiation blockage.

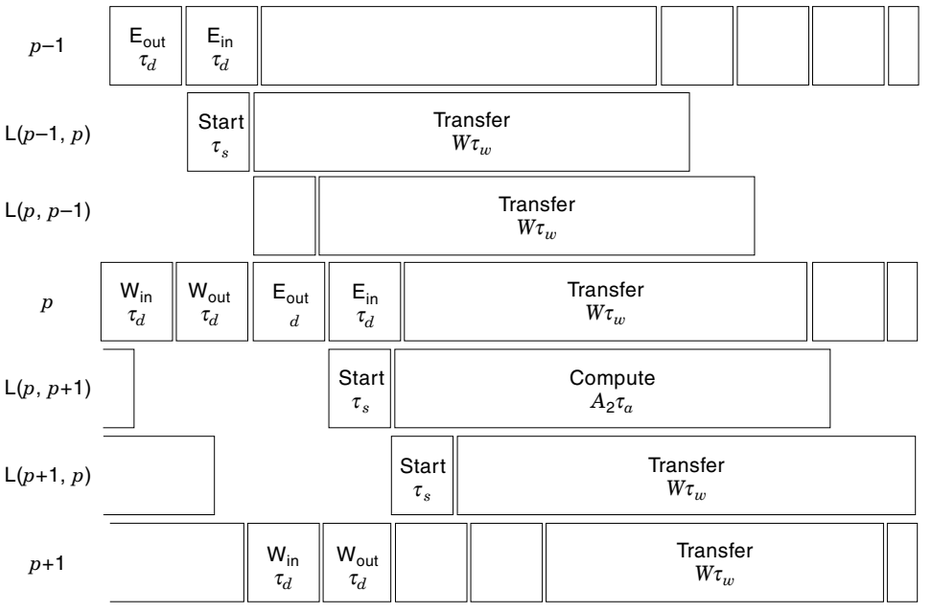

Figure 7. Maximum arithmetic and communications concurrency.

The assignment of the MNR grains to the P processes is dictated by P *arithmetic grain aggregation coefficients* $h_1, h_2, \ldots, h_P$, where

$$\left\lfloor \frac{N}{P} \right\rfloor \le h_p \le \left\lceil \frac{N}{P} \right\rceil, p = 1, 2, \ldots, P, \text{ and } \sum_{p=1}^{P} h_p = N$$

Then the arithmetic grain aggregation coefficients are used to define the *cumulative arithmetic grain aggregation coefficients* $H_0, H_1, \ldots, H_P$, where

$$H_0 = 0, H_p = H_{p-1} + h_p, p = 1, 2, \ldots, P$$

The arithmetic grains assigned to process p for $p = 1, 2, \ldots, P$ are $u^{(r)}_{m,n}$ for all $m = 1, 2, \ldots, M$, for all $n = H_{p-1} + 1, H_{p-1} + 2, \ldots, H_p$, and for all $r = 1, 2, \ldots, R$. The relationship between the discretization grid and the processing array is shown in Fig. 8.

Because the number of arithmetic grains assigned to process p is h_pMR, the number of arithmetic operations executed by process p is $11h_pMR$. For each $r = 1, 2, \ldots, R$, each process p depends on receiving a western boundary of M words consisting of $u^{(*W)}_{m,H_p-1}$ for $m = 1, 2, \ldots, M$ and an eastern boundary of M words consisting of $u^{(*E)}_{m,H_p+1}$ for $m = 1, 2, \ldots, M$. In addition, for each $r = 1, 2, \ldots, R$, each process p must send a western boundary of M words consisting of $u^{(r)}_{m,H_{p-1}+1}$ for $m = 1, 2, \ldots, M$ and an eastern boundary of M words consisting of $u^{(r)}_{m,H_p}$ for $m = 1, 2, \ldots, M$. The total arithmetic and communication complexities for process p are summarized in Table 3.

The order in which arithmetic grains are executed by each process must take into account their interprocess dependencies. Because the GS sweeping dependencies are supersets of the RB dependencies, which in turn are supersets of the J sweeping dependencies, the arithmetic grain ordering for GS sweeping is chosen. For RB sweeping, the indices are relabeled so that all red arithmetic grains precede black arithmetic grains. The execution of arithmetic grain $u^{(r)}_{m,n}$ depends on the input variables given in Table 2. To satisfy these dependencies, the arithmetic grains are executed from top to bottom among rows and from left to right within a row (see Fig. 9).

A *communication grain* is the communication of a single word of boundary information by any process p to the western process $(p - 1)$ or to the eastern process $(p - 1)$. There is an input and output communication grain associated with each arithmetic grain on the left and right edges of Fig. 9. The order in which the communication grains are executed is chosen as the order in which the corresponding boundary information is needed and generated by each process according to the arithmetic grain execution ordering described before. Communication grains between process p and western process $(p - 1)$ are executed from top to bottom, and communication grains between process p and eastern process $(p + 1)$ are also executed from top to bottom.

Let an *arithmetic step* be the contiguous arithmetic grains executed between communications events, and let U be the number of communication grains in any message. The choice of U induces the number of arithmetic grains in each arithmetic step. Because the time to communicate a W-word message is $\tau_c(W) = \tau_s + W\tau_w$, longer messages, that is, large U, result

Table 2. Arithmetic Grain Computation and Communication Complexities

Operations			Input		Output	
+	×	Total	Variables	Words	Variables	Words
6	5	11	$u^{(*N)}_{m-1,n}, u^{(*W)}_{m,n-1}, u^{(r-1)}_{m,n}, u^{(*E)}_{m,n+1}, u^{(*S)}_{m+1,n}$	5	$u^{(r)}_{m,n}$	1

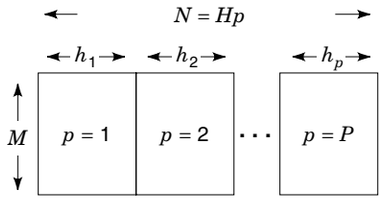

Figure 8. Discretization grid and processing-array relationship.

in a smaller average per word transfer time that reduces overall latency. However, longer messages also contribute to delaying computations on processors that depend on message data; thereby increasing overall latency. Thus expressing latency as a function U affords a means to determine this tradeoff optimally.

The number of input and output words to each process is MR, and therefore, the number of messages to and from each process is given by

$$S = \left\lceil \frac{MR}{U} \right\rceil$$

The reception of each U-word message by process p from process $(p - 1)$ with the corresponding message from process $(p + 1)$ enables computing Uh_p arithmetic grains which generate a pair of U word messages, one needed by process $(p - 1)$ and the other needed by process $(p + 1)$. Thus the execution of Uh_p arithmetic grains is bracketed by communications events which define an arithmetic step and therefore the number of arithmetic steps is S.

Five subroutines common to both concurrent and nonconcurrent parallel implementations of the SOR algorithm are now defined. Each subroutine call of `Comp(s)`, `Wout(s)`, `Eout(s)`, `Win(s)`, and `Ein(s)`, executes approximately $1/S$ of the total of arithmetic grains, western output grains, eastern output grains, western input grains, or eastern input grains, respectively where the argument $s \in [1, 2, \ldots, S]$ specifies which $1/S$ of the total grains is executed for a particular subroutine call. For instance, when $u_{i,j} = u^{(r)}_{m,n}$ with $i = M(r - 1) + m$ and $j = n$, then `Comp(s)` executes $u_{i,j}$ for $i = U(s - 1) + 1, U(s - 1) + 2, \ldots, Us$ and $j = H_{p-1} + 1, H_{p-1} + 2, \ldots, H_p$.

When the algorithm is executed, each process p has computed S arithmetic steps for $s = 1, 2, \ldots, S$, totaling g_pMR grains, sent S messages for $s = 1, 2, \ldots, S$ to process $(p - 1)$, totaling MR words, received S messages for $s = 1, 2, \ldots, S$ messages from process $p - 1$, totaling MR words, sent S messages for $s = 1, 2, \ldots, S$ to process $(p + 1)$, totaling MR words and received S messages for $s = 1, 2, \ldots, S$ from process $(p + 1)$ totaling MR words, excluding the boundary processes $p = 1$ and $p = P$, where the communication to process $(p - 1)$ and $(p + 1)$, respectively, is null.

Table 3. Grain Computation and Communication Complexities for Process p

Arithmetic		Input to Process p, Words		Output from Process p, Words	
Grains	Operations	$p - 1$	$p + 1$	$p - 1$	$p + 1$
h_pMR	$11h_pMR$	MR	MR	MR	MR

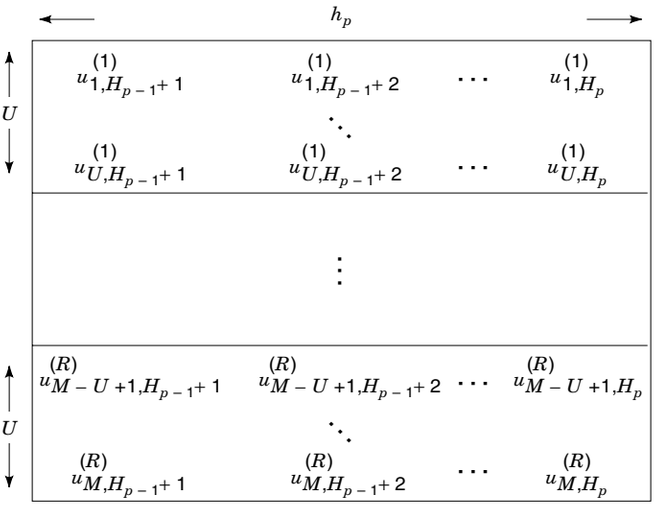

Figure 9. Arithmetic steps for process p.

LATENCY ANALYSIS

In this section, upper bounds on the overall latencies of the parameterized SOR algorithms are quantified for execution on a linear array of processors. In each case, the bounds are computed by evaluating the latency of the process q that has the maximum number of arithmetic grains associated with it, that is, $h_q = \lceil N/P \rceil$, and adding the latency of those processes or portions of processes required before and after execution process q. For convenience, the execution time of an arithmetic grain is denoted τ_g, and thus $\tau_g = 11\tau_a$.

Nonconcurrent SOR

When arithmetic computations and communications cannot be done simultaneously, the algorithm described in Fig. 10 is used for the J and RB sweepings, and the algorithm described in Fig. 11 is used for the GS sweeping, where the parallel execution of instructions $1, 2, \ldots, n$ is indicated by

instruction 1//instruction 2//. . .//instruction n

To satisfy dependency constraints, it is required that $U \leq M$ in the J case, and $U \leq M/2$ in the RB and GS cases.

In the J and RB cases, dependencies allow executing the worst-case process to begin immediately, and then execution proceeds without blocking because its western and eastern neighbors have at most the same number of grains to compute at each arithmetic step. Thus the latency in the J and RB cases is bounded from above by L_{Jn} and L_{RBn} with

$$\begin{aligned} L_{Jn} = L_{RBn} &= S(4(\tau_d + \tau_s + U\tau_w) + h_q U\tau_g) \\ &= \left\lceil \frac{MR}{U} \right\rceil (4\tau_d + 4\tau_s + 4U\tau_w + \lceil N/P \rceil U\tau_g) \end{aligned}$$

```
DO IN PARALLEL FOR p = 1, . . ., P
  for s = 1, 2, . . ., S
    Wout(s)
    Eout(s)
    Win(s)
    Ein(s)
    Comp(s)
  end
END
p = EVEN
```

```
DO IN PARALLEL FOR p = 1, . . ., P
  for s = 1, 2, . . ., S
    Ein(s)
    Win(s)
    Eout(s)
    Wout(s)
    Comp(s)
  end
END
  p = ODD
```

Figure 10. Nonconcurrent 1-D Jacobi/red–black algorithm.

Minimizing over the communication granularity parameter U in the J case gives $U = M$ and latency bound

$$L_{\mathrm{Jn}} = 4R\tau_{\mathrm{d}} + 4R\tau_{\mathrm{s}} + 4MR\tau_{\mathrm{w}} + MR\left\lceil \frac{N}{P} \right\rceil \tau_{\mathrm{g}}$$

and in the RB case gives $U = M/2$ and latency bound

$$L_{RBn} = 8R\tau_{\mathrm{d}} + 8R\tau_{\mathrm{s}} + 4MR\tau_{\mathrm{w}} + MR\left\lceil \frac{N}{P} \right\rceil \tau_{\mathrm{g}}$$

In the nonconcurrent GS case, dependencies block the execution of the worst case process q until processes $p = 1, 2, \ldots, q-1$ have executed their respective first triplet of input communications, arithmetic step, and output communications. Then execution of the worst case process proceeds unblocked because its western and eastern neighbors have at most the same number of grains to compute at each arithmetic step. When the worst case process concludes, processes $p = q+1, q+2, \ldots, P$ must execute their final triplet of input communications, arithmetic step, and output communications. The latency incurred before the process q loop can begin executing is expressed by

$$[1 + 2(q-2)](\tau_{\mathrm{d}} + \tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + \sum_{p=1}^{q-1} h_{\mathrm{p}} U \tau_{\mathrm{g}}$$

the latency incurred executing the process q loop is given by

$$4(S-1)(\tau_{\mathrm{d}} + \tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + (S-1)h_{\mathrm{q}} U \tau_{\mathrm{g}}$$

```
DO IN PARALLEL FOR p = 1, . . ., P
    Wout(1)
    Ein(1)
    for s = 1, 2, . . ., S − 1
      Win(s)
      Winout(s + 1)
      Comp(s)
      Eout(s)
      Ein(s + 1)
    end
    Win(S)
    Comp(S)
    Eout(S)
END
```

Figure 11. Nonconcurrent 1-D Gauss–Seidel algorithm.

and the latency incurred following the process q loop is given by

$$2(P-q)(\tau_{\mathrm{d}} + \tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + \sum_{p=q}^{P} h_{\mathrm{p}} U \tau_{\mathrm{g}}$$

Thus the latency in the GS case is bounded from above by L_{GSn} with

$$L_{\mathrm{GSn}} = \left(4\left\lceil \frac{MR}{U} \right\rceil + 2P - 7\right)(\tau_{\mathrm{d}} + \tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + \left[\left(\left\lceil \frac{MR}{U} \right\rceil - 1\right)\left\lceil \frac{N}{P} \right\rceil U + NU\right]\tau_{\mathrm{g}}$$

Given an SOR algorithm, architectural parameters P, τ_{a}, τ_{d}, τ_{w}, and τ_{s}, and problem parameters M, N, and R, the corresponding latency bound can be plotted as a function of the communication granularity U, and the optimal U may be obtained from the plot. For example, in the nonconcurrent GS case with architectural parameters $P = 8$, $\tau_{\mathrm{a}} = 1.34$ μs, $\tau_{\mathrm{d}} = 120.0$ μs, $\tau_{\mathrm{w}} = 9.0$ μs, and $\tau_{\mathrm{s}} = 12.2$ μs, and problem parameters $M = N = 90$ and $R = 10$, L_{GSn} is plotted as a function of U in Fig. 12. One sees that $U = 1$ yields 669.0 ms for the latency bound and that $U = 20$ yields 240.7 ms, the minimum

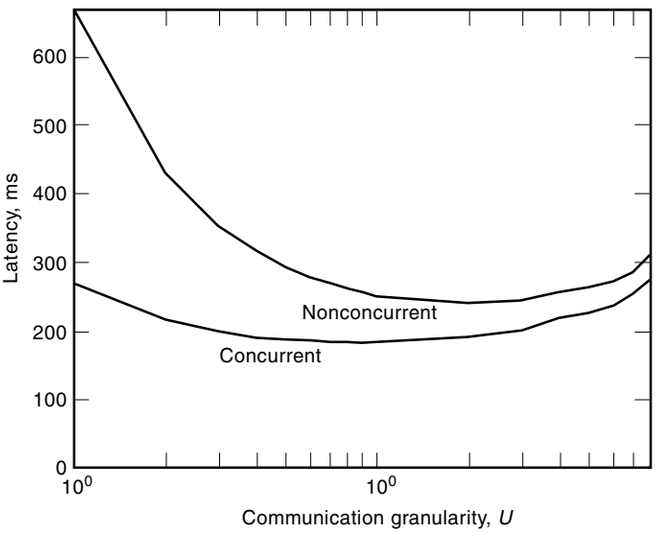

Figure 12. Gauss–Seidel latency vs. communication granularity for $P = 8$, $\tau_{\mathrm{a}} = 1.34$ μs, $\tau_{\mathrm{d}} = 120.0$ μs, $\tau_{\mathrm{w}} = 9.0$ μs, $\tau_{\mathrm{s}} = 12.2$ μs, $M = N = 90$, $R = 10$.

```
DO IN PARALLEL FOR p = 1, . . ., P
  Wout(1) // Eout(1) // Win(1) // Ein(1)
  for s = 1, 2, . . ., S − 1
    Wout(s + 1) // Eout(s + 1) // Win(s + 1) // Ein(s + 1) // Comp(s)
  end
  Comp(S)
END
```

Figure 13. Concurrent 1-D Jacobi/red-black algorithm.

latency bound. This may be compared to the single processor latency of 1193.8 ms, and we may conclude that the use of the customary U produces an efficiency of 22%, and the use of the optimal U produces an efficiency of 62%.

Concurrent SOR

When arithmetic computations and communications can be done simultaneously, the algorithm given in Fig. 13 is used for the J and RB cases, the algorithm given in Fig. 14 is used for the GS case. To satisfy dependency constraints and to permit the concurrent execution of arithmetic grains and communication grains, it is required that $U \le M/2$ in the J case and $U \le M/4$ in the RB and GS cases.

As in the nonconcurrent situation, dependencies in both the J and RB cases, allow execution of the worst case process to begin immediately and to proceed without blocking because its western and eastern neighbors have at most the same number of grains to compute at each arithmetic step. Thus the latency in the J and RB cases is bounded from above by L_{Jc} and L_{RBc} with

$$\begin{aligned} L_{\mathrm{Jc}} = L_{\mathrm{RBc}} &= (\tau_{\mathrm{d}} + \tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) \\ &\quad + (S-1)\max\{\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}, h_q U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\} + h_{\mathrm{q}} U\tau_{\mathrm{g}} \\ &= (\tau_{\mathrm{d}} + \tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + \left(\left\lceil \frac{MR}{U} \right\rceil - 1\right) \\ &\quad \max\left\{\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}, \left\lceil \frac{N}{P} \right\rceil U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\right\} + \left\lceil \frac{N}{P} \right\rceil U\tau_{\mathrm{g}} \end{aligned}$$

If $\tau_{\mathrm{s}} + U\tau_{\mathrm{w}} \le \lceil N/P \rceil (U\tau_{\mathrm{g}} + \tau_{\mathrm{d}})$, then

$$L_{\mathrm{Jc}} = L_{\mathrm{RBc}} = (\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + \left\lceil \frac{MR}{U} \right\rceil \left(U\left\lceil \frac{N}{P} \right\rceil \tau_{\mathrm{g}} + \tau_{\mathrm{d}}\right)$$

In the concurrent GS case, dependencies block the execution of the worst case process q until processes $p = 1, 2, \ldots, q-1$ have executed their respective first triplet of input communications, arithmetic step, and output communications. Then execution of the worst case process proceeds unblocked because its western and eastern neighbors have at most the same number of grains to compute at each arithmetic step. When the worst case process concludes, processes $p = q+1, q+2, \ldots, P$ must execute their final triplet of input communications, arithmetic step, and output communications. The latency incurred before process q can begin executing is expressed by

$$\sum_{p=1}^{q-1} (\tau_{\mathrm{s}} + U\tau_{\mathrm{w}} + \max\{\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}, h_{\mathrm{p}} U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\})$$

the latency incurred executing process q is expressed by

$$(\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + S\max\{\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}, h_{\mathrm{g}} U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\}$$

and the latency incurred following process q is expressed by

$$\sum_{p=q+1}^{P} (\tau_{\mathrm{s}} + U\tau_{\mathrm{w}} + \max\{\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}, h_{\mathrm{p}} U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\})$$

Thus the latency in the GS case is bounded from above by L_{GSc} where

$$\begin{aligned} L_{\mathrm{GSc}} &= \sum_{p=1}^{P} (\tau_{\mathrm{s}} + U\tau_{\mathrm{w}} + \max\{\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}, h_{\mathrm{p}} U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\}) \\ &\quad + (S-1)\max\{\tau_{\mathrm{s}} + U\tau_{\mathrm{w}}, \left\lceil \frac{N}{P} \right\rceil U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\} \end{aligned}$$

If $\tau_{\mathrm{s}} + \mathrm{U}\tau_{\mathrm{w}} \le h_{\mathrm{p}}\, U\tau_{\mathrm{g}}$ for all p, then

$$\begin{aligned} L_{\mathrm{GSc}} &= P(\tau_{\mathrm{d}} + \tau_{\mathrm{s}} + U\tau_{\mathrm{w}}) + \left(\left\lceil \frac{MR}{U} \right\rceil - 1\right) \left(\left\lceil \frac{N}{P} \right\rceil U\tau_{\mathrm{g}} + \tau_{\mathrm{d}}\right) \\ &\quad + NU\tau_{\mathrm{g}} \end{aligned}$$

```
DO IN PARALLEL FOR p = 1, . . ., P
  Wout(1)
  Wout(2) // Ein(1)
  Wout(3) // Win(1)
  Wout(4) // Win(2) // Ein(2) // Comp(1)
  for s = 2, 3, . . ., S − 3
    Wout(s + 3) // Win(s + 1) // Ein(s + 1) // Comp(s) // Eout(s − 1)
  end
  Win(S − 1) // Ein(S − 1) // Comp(S − 2) // Eout(S − 3)
  Win(S) // Ein(S) // Comp(S − 1) // Eout(S − 2)
  Comp(S) // Eout(S − 1)
  Eout(S)
END
```

Figure 14. Concurrent 1-D Gauss–Seidel algorithm.

Once again, given architectural parameters P, τ_a, τ_d, τ_w, and τ_s, and problem parameters M, N, and R, the corresponding latency bound can be plotted as a function of the communication granularity U, and the optimal U may be obtained from the plot. For example, in the concurrent GS case with architectural parameters $P = 8$, $\tau_a = 1.34$ μs, $\tau_d = 120.0$ μs, $\tau_w = 9.0$ μs, and $\tau_s = 12.2$ μs, and problem parameters $M = N = 90$ and $R = 10$, L_{GSc} is plotted as a function of U in Fig. 12. One sees that $U = 1$ yields 269.0 ms for the latency bound and that $U = 9$ yields 183.5 ms, the minimum latency bound. This may be compared to the single processor latency of 1193.8 ms, and we may conclude that the use of the customary U produces an efficiency of 55% and the use of the optimal U produces an efficiency of 81%.

CONCLUSIONS

Whenever a W-word message has to be transferred from one processor to another, one incurs a computational cost τ_d to initiate and synchronize the message transfer and also a communication link cost $\tau_c(W) = \tau_s + W\tau_w$. It follows that longer messages result in a smaller average per word computational overhead and a smaller average per word communication transfer time that reduces overall latency. However, longer messages delay computations on processors which depend on message data, increasing latency. Using parameterized algorithms in which message size can be adjusted allows balancing message overhead against delays due to computational dependencies. Then, expressing latency as a function of *communication granularity,* which is related to message length, allows the optimally determining the necessary tradeoff. Parameterizing algorithms also has the advantage that high efficiencies are more easily maintained when hosted on a multiplicity of architectures because parameters may be adjusted for each architecture.

In this article, it has been shown that the relationship between latency and communication granularity U for a family of parametrized parallel SOR algorithms is pronounced and that the reduction in latency with an optimal choice of U is significant. The efficiencies of these algorithms are high whenever the corresponding optimal communication granularity is used, suggesting that architectures which are more complicated than the linear array need not be considered. Given a problem, one can determine or estimate the number of iterations R_J, R_{RB}, and R_{GS} required to achieve a desired accuracy for each sweeping order J, RB, and GS, find the optimal U and the corresponding latency for each case, and then choose the best SOR algorithm. The GS sweeping order is of the most interest, however, because it has a generally superior rate of convergence and because of its amenability to the enhancements mentioned in the introduction.

BIBLIOGRAPHY

1. R. E. Boisvert, Algorithms for special tridiagonal systems, *SIAM J. Sci. Stat. Comput.,* **12**: 423–442, 1991.
2. C. J. Ribbens, L. T. Watson, and C. Desa, Toward parallel mathematical software for elliptic partial differential equations, *ACM Trans. Math. Softw.,* **19**: 457–473, 1993.
3. G. Rodrigue, *Parallel Processing for Scientific Computations,* Philadelphia: SIAM, 1989.
4. H. A. Van der Vorst, High performance preconditioning, *SIAM J. Sci. Stat. Comput.,* **10**: 1174–1185, 1989.
5. A. Asenov, D. Reid, and J. R. Barker, Speed-up of scalable iterative linear solvers implemented on an array of transputers, *Parallel Comput.,* **20**: 375–387, 1994.
6. N. H. Naik and J. Van Rosendale, The improved robustness of multigrid elliptic solvers based on multiple semicoarsened grids, *SIAM J. Numer. Anal.,* **30**: 215–229, 1993.
7. W. H. Press et al., *Numerical Recipes in C,* Cambridge Univ. Press, 1992, Chap. 19.
8. L. M. Adams and H. F. Jordan, Is SOR color-blind?, *J. Sci. Stat. Comput.,* **7**: 490–506, 1986.
9. C.-C. J. Kuo, T. F. Chan, and C. Tong, Two color Fourier analysis of iterative algorithms for elliptic problems with red/black ordering, *SIAM J. Sci. Stat. Comput.,* **11**: 767–794, 1990.
10. C.-C. J. Kuo and B. C. Levy, Discretization and solution of elliptic PDEs—a digital signal processing approach, *Proc. IEEE,* **12**: 1808–1842, 1990.
11. J. M. Ortega and R. G. Voigt, Solution of partial differential equations on vector and parallel computers, *SIAM Rev.,* **27**: 149–240, 1985.
12. R. S. Varga, *Matrix Iterative Analysis,* Englewood Cliffs, NJ: Prentice-Hall, 1962.
13. L. A. Hageman and D. M. Young, *Applied Iterative Methods,* New York: Academic Press, 1981.
14. E. F. Botta and A. E. P. Veldman, On local relaxation methods and their application to convection-diffusion equations, *J. Comput. Phys.,* **48**: 127–149, 1981.
15. L. W. Ehrlich, An ad hoc SOR method, *J. Comput. Phys.,* **44**: 31–45, 1981.
16. C.- C. J. Kuo, B. C. Levy, and B. R. Musicus, A local relaxation method for solving elliptic PDEs on mesh-connected arrays, *SIAM J. Sci. Stat. Comput.,* **8**: 550–573, 1987.
17. R. Biswas, J. E. Flaherty, and M. Benantar, Advances in adaptive parallel processing for field applications, *IEEE Trans. Magn.,* **27**: 3768–3773, 1991.
18. D. P. O'Leary and P. Whitman, Parallel QR factorization by Householder and modified Gram–Schmidt algorithms, *Parallel Comput.,* **16**: 99–112, 1990.
19. G. G. L. Meyer and M. Pascale, A family of Parallel QR Factorization Algorithms, *High Performance Comput. Symp. '95,* 1995, pp. 95–106.
20. M. A. Pirozzi, The fast numerical solution of mildly nonlinear elliptic boundary value problems on multiprocessors, *Parallel Comput.,* **19**: 1117–1128, 1993.

GERARD G. L. MEYER
Johns Hopkins University
MICHAEL V. PASCALE
Northrop Grumman