Scalability analysis of parallel GMRES implementations

Maria Sosonkina
Department of Computer Science
University of Minnesota
320 Heller Hall, 10 University Drive,
Duluth, MN 55812
masha@d.umn.edu

Donald C. S. Allison and
Layne T. Watson
Department of Computer Science
Virginia Polytechnic Institute and
State University
Blacksburg, VA 24061
{allison, ltw}@cs.vt.edu

Abstract

Applications involving large sparse nonsymmetric linear systems encourage parallel implementations of robust iterative solution methods, such as GMRES(k). Two parallel versions of GMRES(k) based on different data distributions and using Householder reflections in the orthogonalization phase, and variations of these which adapt the restart value k, are analyzed with respect to scalability (their ability to maintain fixed efficiency with an increase in problem size and number of processors). A theoretical algorithm-machine model for scalability is derived and validated by experiments on three parallel computers, each with different machine characteristics.

1. Introduction

For large scale problems, the task of writing efficient parallel linear system solvers is particularly important. These problems usually involve sparse linear systems and require general purpose iterative methods. These methods preserve the sparsity of matrices and do not involve complete matrix factorization. A variation of the popular linear system solution tool GMRES [13] is considered in this paper. This variation [17] uses an adaptive strategy to deal with varying difficulties of linear systems which are to be solved by a nonlinear algorithm. For difficult structural mechanics problems, the implementation of this version on a sequential machine is superior to other GMRES variants that have no adaptive capabilities and use different orthogonalization schemes. Therefore, this adaptive version of GMRES (precisely, its restarted version GMRES(k)) has been considered for parallel implementation and parallel performance analysis.

Two variations of parallel GMRES(k) are considered here. Both adapt the restart value k and use Householder reflections in the orthogonalization phase in order to achieve high accuracy. One variation uses a fixed assignment of the rows of the coefficient matrix to the processors, while the other uses a sophisticated graph partitioning algorithm to assign rows to processors.

The implementation of several restart cycles of these algorithms on a parallel computer is analyzed using the isoefficiency metric [7]. Despite being rather general, this metric provides valuable insights into implementation scalability and its relationship with important machine parameters, which are involved in the construction of an isoefficiency model. This paper presents the validation of the constructed model by carrying out experiments on three parallel computers with differing speed, memory organization, input/output capabilities, and network interconnections. The Intel Paragon is a relatively slow machine with a small cache, a centralized I/O mechanism, and a 2-D mesh interconnecting network. The IBM SP2 and the Cray T3E are both an order of magnitude faster, have different memory configurations, and have interconnections with more uniform latencies. The primary thrust of this paper is to execute a theoretical isoefficiency analysis for these algorithm/machine combinations and then validate it by experiment.

Section 2 gives a brief description of the adaptive GMRES(k) algorithm and its parallel implementations using Householder reflections. A derivation of the isoefficiency function for restarted GMRES is presented in §3 followed by numerical experiments in §4. Section 5 contains conclusions.

2. Adaptive GMRES algorithm

The numerical linear algebra terminology used here is standard; see, for example, the textbook by Saad [12]. The GMRES algorithm is used to solve a linear system $Ax = b$ with an $n \times n$ nonsymmetric invertible coefficient matrix A. Similar to the classical conjugate gradient method, GMRES produces approximate solutions x_j which are characterized by a minimization property over the Krylov subspaces $K(j, A, r_0) \equiv \text{span}\{r_0,\ Ar_0,\ A^2r_0,\ \cdots,\ A^{(j-1)}r_0\}$, where $r_0 = b - Ax_0$ and j is the iteration number. However, unlike the conjugate gradient algorithm, the work and memory required by GMRES grow proportionately to the iteration number since GMRES needs all j vectors to construct an orthonormal basis for $K(j, A, r_0)$. In practice, a restarted version GMRES(k) is used, where the algorithm is restarted every k iterations. GMRES(k) takes x_k as the initial guess for the next cycle of k iterations, and continues until the residual norm is small enough.

The disadvantage of the restarted version is that it may stagnate and never reach the solution. The essence of the adaptive GMRES strategy is to adapt the parameter k to the problem, in the same way a variable order ODE algorithm tunes the order k. With modern programming languages, which provide pointers and dynamic memory management, dealing with the variable storage requirements implied by varying k is not difficult. A test of stagnation developed in [3] detects an insufficient residual norm reduction in the restart number k of steps by estimating the GMRES behavior on a particular linear system. Slow progress of GMRES(k), indicating that an increase in the restart value k may be beneficial [18], can be detected with a similar test.

The convergence of GMRES may also be seriously affected by roundoff error, which is especially noticeable when a high accuracy solution is required. When the orthogonalization phase of GMRES is performed by the modified Gram-Schmidt process, GMRES is susceptible to numerical instability. In practice, the reorthogonalization phase often complements modified Gram-Schmidt to benefit stability as shown in [4] and [10]. However, for difficult structural mechanics problems such as those described in [17], the reorthogonalization produced no improvements. Therefore Householder reflections were adopted in the orthogonalization phase. As shown in [1], the orthogonalization with Householder reflections is more robust than the modified Gram-Schmidt process. In theory, the implementation of GMRES using Householder reflections is about twice as expensive as when modified Gram-Schmidt is used [19]. However, the Householder reflection method produces a more accurate orthogonalization of the Krylov subspace basis when the basis vectors are nearly linearly dependent and the modified Gram-Schmidt method fails to orthogonalize the basis vectors; this can result in fewer GMRES iterations compensating for the higher cost per iteration using Householder reflections. GMRES(k) may exceed an iteration limit when it is affected by roundoff errors in the case of a (nearly) singular GMRES least-squares problem. The condition number of the GMRES least-squares problem is monitored by the incremental condition estimate [2] as in [3]. GMRES(k) aborts when the estimated condition number is greater than some large number, e.g., $1/(50\mathbf{u})$, where $\mathbf{u}$ is the machine unit round off. Pseudocode for an adaptive version of GMRES(k) with orthogonalization via Householder reflections (as in [19]) is given in [17].

In parallel environments, the choice of the orthogonalization process for the Krylov subspace basis vectors depends not only on the accuracy of the process but also on the amount and type of global communication it incurs. For some orthogonalization procedures, only one of the two

requirements is satisfied. For example, in serial implementations of the GMRES method, the modified version of the Gram-Schmidt process is often used as being sufficiently accurate for a number of problems. In parallel GMRES implementations, however, the modified Gram-Schmidt process exhibits a large communication overhead. Because of this and the need for high accuracy, here an implementation of the Householder reflection orthogonalization in GMRES(k) proposed in [19] is adapted to work in parallel. The parallel version employs an algorithm developed in [14] for generating and applying Householder reflections. This algorithm avoids dot-products and all-to-all communications. Pseudocode for the algorithm generating Householder reflections (called HG) at the jth GMRES(k) iteration on the processor $proc$ (in a ring of p processors) is given in Figure 1 (top). It is followed by the pseudocode of the application of Householder reflections in Figure 1 (bottom).

```
if (proc = 1) then s := j else s := 0 end if
determine H_{s+1} such that H_{s+1} v_loc ≡ w_loc has zeros
  after the (s + 1)st component;
if (proc = 1) then
      send w_loc(s + 1) to right;
else
      receive w from left;
      determine G_1 such that w_loc(1) = 0; update w;
      if (proc ≠ p) send w to right;
end if
```

```
for i := j downto 1 step -1
    if (proc = 1) then
          send w_loc(i) to p;
          receive w_loc(i) from right;
          sc := i;
    else
          receive w from right;
          apply G_1 to (w, w_loc(1));
          send w to left;
          sc := 1;
    end if
    apply H_sc to w_loc(sc :);
end for
```

Figure 1. Parallel Householder reflection generation (top) and application (bottom).

In Figure 1, H and G denote the Householder transformation matrix and the Givens rotation matrix (as given in [6]), respectively; v_{loc} denotes a portion of the Krylov subspace vector $A^j r_0$ located on a processor; p, $left$, $right$ are the processors with the highest rank, with the $proc - 1$ rank, and the $proc + 1$ rank, respectively. It is also assumed that the first processor has the jth row of the input matrix. However, the design presented in Figure 1 admits only a special case of the matrix row distribution: assignment of a block of contiguous rows to each processor (called block-striped distribution), which is rarely advantageous for an arbitrary unstructured matrix. For unstructured matrices a graph partitioning is used to minimize the communication to computation

ratio. In the current implementation, a graph partitioning algorithm from the MeTiS package [9] is used to partition the input matrix by rows, and the parallel version of the matrix-vector multiply is performed as in [11]. The matrix-vector product requires that the components of all vectors are distributed in accordance with the corresponding matrix rows and allows overlapping of computation and communication.

```
if (j = 1) then
    s := 1;
else
    if (proc has (j − 1)st row) then s := s + 1;
end if
determine H_s such that H_s v_loc ≡ w_loc has zeros
  after the sth component;
if (proc has jth row) then
    ring_end := left;
    send w_loc(s + 1) and ring_end to right;
else
    receive w and ring_end from left;
    determine G_s such that w_loc(s) = 0; update w;
    if (proc ≠ ring_end) then send w to right;
end if
```

```
sc := s;
for i := j downto 1 step -1
    if (proc has ith row) then
        if (sc ≠ 1) then sc := sc − 1;
        send w_loc(sc) to left;
        receive w_loc(sc) from right;
    else
        receive w from right;
        if (G_sc exists) then
          apply G_sc to (w, w_loc(sc));
        end if
        send w to left;
    end if
    apply H_sc to w_loc(sc :);
end for
```

Figure 2. Modified Householder reflection generation (top) and application (bottom).

To use the algorithms in Figure 1, the redistribution of a vector requires $\mathcal{O}(p^2)$ communications at each GMRES(k) iteration, which is highly impractical and reduces the efficiency gained by the distributed matrix-vector product. Thus, it is beneficial to develop an extension (called MHG) of the algorithms in Figure 1 which accepts an arbitrary row distribution among processors. Figure 2 (top) shows the pseudocode for MHG, where the row indexing refers to the original matrix before graph partitioning. Usually, the subspace dimension is much smaller than the matrix dimension

and the graph partitioning algorithm produces a balanced workload by assigning an almost equal number of rows to each processor.

Thus, the case when the index s within v_{loc} becomes equal to the size of a local partition (size of v_{loc}) occurs rarely for large matrices, unless the number of processors is very large.

3. Comparative scalability analysis

An algorithm-architecture scalability analysis estimates, in a single expression, characteristics of the algorithm as well as parameters of the architecture on which the algorithm is implemented. Thus, testing an algorithm on a small number of processors allows one to predict its performance on a larger number of processors. The following terminology is needed to support the proposed scalability analysis. A *parallel system* is a parallel algorithm and machine combination. The *useful work* W is the total number of basic floating point operations required to solve the problem. W depends on the *problem size* $\bar{N}$, which is a vector of problem-specific parameters such as problem dimensions and the number of nonzero entries in a matrix. For numerical linear algebra problems solved by iterative methods, $\bar{N}$ may also include an indication of problem difficulty, such as the Krylov subspace dimension k used in GMRES(k) or the condition number. In general, choosing $\bar{N}$ requires a detailed investigation of a given problem and of the way scaling of problem dimensions affects the increase in W [15].

The *sequential execution time* T_1 characterizes the time interval needed to solve a given problem on a uniprocessor. If the time of executing an integer operation is negligible compared with the time t_c of performing a floating point operation and if T_1 is spent in useful work only, then $T_1 = t_c W$. An assumption here is that T_1 is spent doing useful work only. The *parallel execution time* T_p is the time taken by p processors to solve the problem. The *total parallel overhead* V is the sum of all overheads incurred by all the processors during parallel execution of the algorithm. For a given parallel system, V is a function of the useful work W and number of processors p: $V = pT_p(W,p) - T_1(W)$. The *efficiency* E is the ratio of the speedup $S(p)$ to p, where the speedup $S(p) = T_1/T_p$. Hence, $E = \frac{T_1}{pT_p} = \frac{1}{(1+V/T_1)}$. A parallel system is called *scalable* if $V = \mathcal{O}(W)$ and unscalable otherwise. For scalable systems, it is possible to keep the efficiency fixed and to monitor the rate of increase in W and p with the *isoefficiency function* $f_E(p)$, as proposed in [7]. The isoefficiency function is defined implicitly by the following relation between W and the parallel overhead V:

$$W(\bar{N}) = eV(W(\bar{N}), p, \bar{h}), \tag{1}$$

where $e = \frac{E}{t_c(1-E)}$ is a constant and $\bar{h}$ is a vector of machine-dependent parameters affecting the amount of the parallel overhead. Usually, the communication cost of the total parallel overhead incorporates these parameters, which are defined in accordance with a communication model supported by a given architecture.

3.1. Operation count for the useful work

Let N be the scaled matrix dimension and N_z be the number of nonzeros in the scaled matrix. To obtain an operation count for the useful work in GMRES(k) at the jth iteration the following parts of the pseudocode for GMRES(k) can be distinguished: (a) matrix-vector product, which takes N_z operations; (b) Householder reflection generation, which takes $2(N-j+1)$ operations; (c) Householder reflection application, which requires $4(N-i+1)$ operations, $i = 1, \ldots, j$. The remaining work is accomplished in $\mathcal{O}(k)$ operations. For a single restart cycle, the useful work

$$W \approx (k+1)N_z + N(2k^2 + 5k + 3) + C_k, \tag{2}$$

where C_k is a term depending only on k.

3.2. Description of the test problem

The scalability analysis is performed on a real-world problem representing a commercial circuit design at AT&T Bell Laboratories. The dimension of this problem is $n = 125$ and the number of nonzeros $n_z = 782$. The matrix $A = \{a_{ij}\}$, $i, j = 1, \ldots n$ of the linear system is unsymmetric, has no particular structure, and 88% of its rows are weakly diagonally dominant. (Weak diagonal dominance is defined as $a_{ii} \geq \sum_j a_{ij}$.) Here scaling the problem K times means assembling K replicas of the $n \times n$ matrix of coefficients in an $N \times N$ matrix, where $N = K \times n$ and the number of nonzeros $N_z = K \times n_z$, as shown in Figure 3 for $K = 5$. Both the initial solution vector and right-hand side are $e_N = (0, \ldots, 0, 1)^t$.

The choice of the test problem affects only a part of the scalability analysis of GMRES(k), namely, matrix-vector multiplication. The GMRES acceleration remains independent of the test problem. Therefore the analysis presented in this paper can be easily extended to any test problem as long as the parallel overhead for its matrix-vector multiply is known.

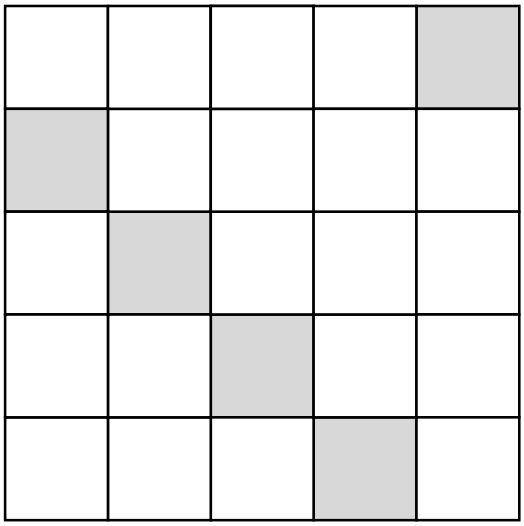

Figure 3. Scaling of problem size.

3.3. Derivation of the isoefficiency function

When GMRES(k) is used for solving the test problem, Equation (1) takes the following form:

$$W(Kn, k) = eV(W(Kn, k), p, \bar{h}), \tag{3}$$

where $\bar{h} = (t_s, t_w)^t$ is a vector of the hardware-dependent communication characteristics: start-up time t_s and transmission time t_w. Parallel GMRES(k) with Householder reflection orthogonalization incurs an overhead in matrix-vector product, Householder reflection generation and application, and in the residual norm update, which is performed on a single processor.

The isoefficiency function is derived under the following assumptions for the serial time calculation, graph partitioning of a matrix, and communication handling: (a) all the work in the sequential algorithm is considered useful; (b) graph partitioning algorithm produces balanced partitions; (c) for large scale problems, partitions produced by graph partitioning keep the computation

to communication ratio no smaller than block-striped partitioning; (d) each processor holds Kn/p rows of the matrix (Figure 3), where $p \leq K$; $Kn/p \geq n$) (e) if $p \leq K$, then each $n \times n$ matrix block is partitioned between no more than two processors; (f) in matrix-vector product, communication is performed by asynchronous sends and synchronous receives; (g) the time complexity of the MPI broadcast operation implemented on the Intel Paragon, IBM SP2, and Cray T3E does not grow substantially as the number of processors increases. Note that, since for the majority of realistic applications the amount of computation grows superlinearly in the number of processors, assumption (d) is not very constraining and is satisfied using either block-striped partitioning or an algorithm from MeTiS. Also, assumption (g) stems from wormhole routing schemes used in the given parallel architectures.

The parallel overhead due to a matrix-vector product can be predicted by considering the nonzero structure of a given coefficient matrix. For the type of matrices shown in Figure 3, a processor receives at most n vector components from no more than $\lceil p/K \rceil$ processors and sends its Kn/p vector components to at most $\lceil p/K \rceil$ processors. Let $V_{r_j}^{MV}$ and $V_{s_j}^{MV}$ be the total overheads incurred by the processors at data receiving and sending stages of a matrix-vector product, respectively. Then the total parallel overhead of the matrix-vector product at the jth iteration is $V_j^{MV} = V_{r_j}^{MV} + V_{s_j}^{MV}$ with

$$V_{r_j}^{MV} \approx \left(\left\lceil \frac{p}{K} \right\rceil t_s + nt_w + C_0 \right) \times p \qquad \text{and} \qquad V_{s_j}^{MV} \approx \left(\left\lceil \frac{p}{K} \right\rceil t_s + \frac{Kn}{p} t_w - C_1 \right) \times p,$$

where the constants C_0 and C_1 describe the waiting and communication-computation overlapping times for asynchronous communications, respectively. For the problem sizes considered here, $Kn/p \geq n$, and thus $\lceil p/K \rceil \, t_s \leq t_s$. Combining $V_{r_j}^{MV}$ and $V_{s_j}^{MV}$ yields

$$V_j^{MV} \approx \left(2t_s + \left(n + \frac{Kn}{p} \right) t_w + C_0 - C_1 \right) \times p. \tag{4}$$

Observe that $C_0 - C_1 \approx 0$ since the waiting time of each processor during asynchronous matrix vector product is compensated by the time gain in communication-computation overlapping while sending the information. Thus, $C_0 - C_1$ will be dropped from the expression for V_j^{MV}.

Householder reflection generation and application cause a noticeable communication and nonessential work overhead. At the jth GMRES(k) iteration, the overhead V_j^H due to Householder reflection generation and application comprises the overheads $V_{a_j}^H$ and $V_{c_j}^H$ caused by applying and generating Householder reflections, respectively, such that $V_j^H = 2V_{a_j}^H + V_{c_j}^H$ with

$$V_{a_j}^H \approx \Big[jp\big(2(t_s + t_w) + g_a\big) \Big] \times p \qquad \text{and} \qquad V_{c_j}^H \approx \Big[p\big(2(t_s + t_w) + g_c\big) \Big] \times p, \tag{5}$$

where g_c is the number of operations needed to create a Givens rotation and g_a is the number of operations needed to perform a Givens rotation to zero out one vector component. Since Householder reflections are applied twice per GMRES iteration, $V_{a_j}^H$ has a coefficient of two.

Another source of the parallel overhead appears in estimating the condition number of the GMRES least squares problem, which is done on a single processor. For a typical case, when the Krylov subspace dimension is $j, j = 1, \ldots, k$, gathering $\mathcal{O}(j)$ vector components on a processor, estimating the condition number incrementally, and updating the residual norm (via a broadcast

operation) are relatively inexpensive, since the subspace dimension is much smaller than the matrix dimension N for large scale problems. Global all-to-all communication would be required to perform the condition number estimation in parallel. The parallel overhead $V_j^{IC} = \mathcal{O}(pj)$, which is caused by the time spent to gather $\mathcal{O}(j)$ values, to update the residual norm, and to perform $(j-1)$ operations of the condition number estimation. At the jth GMRES(k) iteration, the parallel overhead V_j^{IC} is caused by the time spent to exchange $\mathcal{O}(j)$ values, to update the residual norm by application of j previous Givens rotations and by generation of a new Givens rotation, and to perform approximately $(j-1)$ operations of the incremental condition number estimation. Thus,

$$V_j^{IC} \approx \Big[2j(t_s + t_w) + jg_a + g_c + (j-1)\Big] \times p. \tag{6}$$

The total parallel overhead V_j ($V_j = V_j^{MV} + V_j^H + V_j^{IC}$) incurred at the jth iteration of the GMRES(k) algorithm with Householder reflections in its orthogonalization stage is

$$\begin{aligned} V_j \approx & \Big[2t_s + \Big(n + \frac{Kn}{p}\Big)t_w\Big]p + \Big[2jp\Big(2(t_s + t_w) + g_a\Big) + p\Big(2(t_s + t_w) + g_c\Big)\Big]p \\ & + \Big[2j(t_s + t_w) + jg_a + g_c + (j-1)\Big]p. \end{aligned}$$

When a restart cycle is finished, i.e., $j = k$, the GMRES(k) algorithm has performed k matrix-vector products, $k+1$ Householder reflection generations, k residual norm updates along with k incremental condition number estimates, and $2k$ Householder reflection applications. At the end of a restart, GMRES(k) calculates the true residual norm using one more matrix-vector product and one more Householder reflection application to correct the current solution. Combining all the overhead terms incurred during k iterations, the expression for the total overhead is

$$V = \sum_{j=1}^{k+1}\Big(V_j^{MV} + V_{c_j}^H\Big) + \sum_{j=1}^{k}\Big(2V_{a_j}^H + V_j^{IC}\Big) + V_{a_k}^H.$$

Substituting equations (4), (5), and (6) into this equation results in

$$V \approx C_g p^2 + (\bar{h}^t\bar{c} + C'_k)p + Kn(k+1)t_w,$$

where

$$\begin{aligned} C_g &= k(k+2)\Big(2(t_s + t_w) + g_a\Big) + (k+1)\Big(2(t_s + t_w) + g_c\Big), \\ C'_k &= k\left(\frac{(k+1)}{2}g_a + \frac{(k-1)}{2} + g_c\right), \end{aligned}$$

and $\bar{h} = (t_s, t_w)^t$, $\bar{c} = (c_1, c_2)^t$ with $c_1 = k^2 + 3k + 2$ and $c_2 = k^2 + (n+1)k + n$. The expression for V is a quadratic polynomial in p. Thus for a fixed k, the fastest growing term is the leading term. The leading term in V comes from creating and applying Givens rotations on processors logically connected in a ring. In the test problem considered here, $N_z \approx 6Kn$. Thus $W \approx Kn(2k^2 + 11k + 9) + C_k$. By substituting the expressions for V and W into equation (3), the relation can be derived for K:

$$K \approx \ e \times \Big[\frac{C_g}{n\,(2k^2 + 11k + 9 - e(k+1)t_w)}p^2 + \frac{\bar{h}^t\bar{c} + C'_k}{n\,(2k^2 + 11k + 9 - e(k+1)t_w)}p\Big]. \tag{7}$$

Note that since the term containing C_k is small and has no dependence on either K or p, it does not appear in equation (7).

If one considers the GMRES(k) solution process as consisting of l restart cycles and including the computation of the initial residual r_0 (which requires one matrix-vector product and one subtraction), then equation (7) can be rewritten as [16]

$$l \times Kn\Big[(2k^2 + 11k + 9) + C_k - e(k+1)t_w\Big] + Kn(7 - et_w) \approx \; l \times e \times \Big[C_g p^2 + (\bar{h}^t \bar{c} + C'_k)p\Big].$$

4. Numerical Experiments

The behavior of six different parallel systems has been studied. An algorithm component of each parallel system is either GMRES(k) with HG orthogonalization or GMRES(k) with MHG orthogonalization, denoted by HG_ and MHG_, respectively. An architecture component is one of three parallel computers: IBM SP2, Cray T3E, or Intel Paragon, denoted by the characters S, T, or P, respectively, appended to HG_ and MHG_. To examine the isoefficiency scalability of these parallel systems, the (processor number, problem size) pairs with the same efficiency were selected. For the efficiency computation, the parallel time needed to perform two GMRES(k) restart cycles and the initial residual r_0 computation by a parallel system was recorded as well as the time for executing the same algorithm on a uniprocessor.

4.1. Machine-dependent constants

To calibrate the expression of the isoefficiency function, the constants t_c, t_s, and t_w are determined for the target parallel computers. Their numerical values are obtained using simple appropriate models, for which a sufficient amount of empirical data can be collected to estimate accurately these values. The time t_c needed to perform a floating point operation is obtained as the parameter of a linear regression model $T_1 = t_c W'$, where W' includes an operation count for W and additional terms capturing the effects of memory hierarchy operations when solving large scale problems by a sequential algorithm. Clearly, the value of W' differs from one parallel architecture to another depending on such factors as the memory size and the interface mechanism among memory layers and processor. The amount of time that accounts for the memory hierarchy operations can often be modeled only by studying particular cases of a problem solved on a given architecture. On supercomputers with small cache and memory sizes, such as the Intel Paragon (16KB of level-one on-board data cache), memory access and paging operations affect the overall computation time considerably. In a given sequential algorithm, this effect is already noticeable for medium-size ($N \approx 5000$) problems. Thus the cost of memory accesses (the cost of load/store operations) to compute a floating point value is added to the overall number of operations performed by the sequential algorithm. To perform a floating point operation, two loads from slow memory and one store operation are needed in the worst case.

Observation of the performance of the sequential algorithm on the IBM SP2 also suggests that a portion of its execution time is spent on fetching data from a slower memory. Although an IBM SP2 processor has a significant amount of data cache (64KB and 256KB for Models 390 and 590, respectively), this cache is not placed on chip, thus the interface with the processor presents a bandwidth bottleneck. Also, neither Model 390 nor Model 590 has level-two cache, which plays an important role in floating point computations. Solution of large scientific problems on the IBM SP2 causes a high cache miss rate and, subsequently, frequent references to main memory, which can significantly affect the execution time. Such an effect is especially pronounced

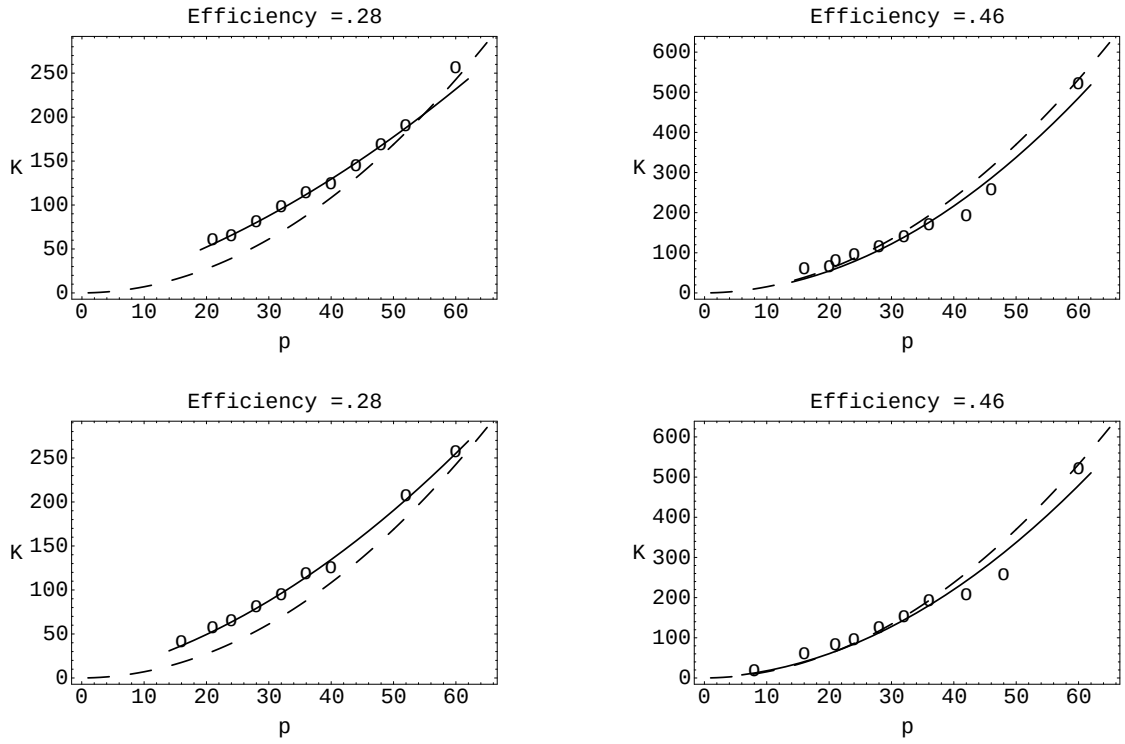

Figure 4. Isoefficiency curves for HG_P (k = 15, top) and MHG_P (k = 15, bottom). Dashed line — theoretical; solid line — fit to data. The vertical axis is the scaling factor K; the horizontal axis is the number of processors p.

in architectures without a two-level cache such as the IBM SP2 Models 390 and 590. Similarly to the Intel Paragon, the useful work W on the IBM SP2 was augmented by the cost of the load and store operations. The total work of a sequential algorithm W' is approximately equal to $3\Big[(k+1)N_z + N(2k^2+5k+3) + C_k\Big]$ on the Intel Paragon and IBM SP2.

On the other hand, each processor of the Cray T3E is coupled with 8KB of data cache and 96KB of level-two data cache. In addition, there are another 4MB of on-board data cache. This is enough cache capacity to hold all the floating point data of the problem sizes considered here with a high cache hit rate. Thus, $W' \approx W$ is acceptable in this case.

The communication start-up time and the transmission time of a double precision number are determined from a communication cost model $T_{comm} = t_s + t_w L$, where T_{comm} is the response variable and L is the predictor variable. This model estimates the communication time T_{comm} between two processors under the assumption that the time required to send a message from one processor to another is independent of processor location and the number of other processors that might be communicating at the same time. The experimental data were gathered from measuring the time T'_{comm} needed to exchange a message between two processors ($T'_{comm} = 2T_{comm}$).

For the IBM SP2, the linear regression $T_1 = 0.024W'$ models the uniprocessor CPU time with the standard deviation of errors equal to 1.84. The regression explains 81% of the variation in T_1, because the coefficient of determination of this model is 0.81. For the linear regression $T_{comm} = 69 + 0.45L$, the standard deviation of errors is 9.6 and the coefficient of determination is 0.97.

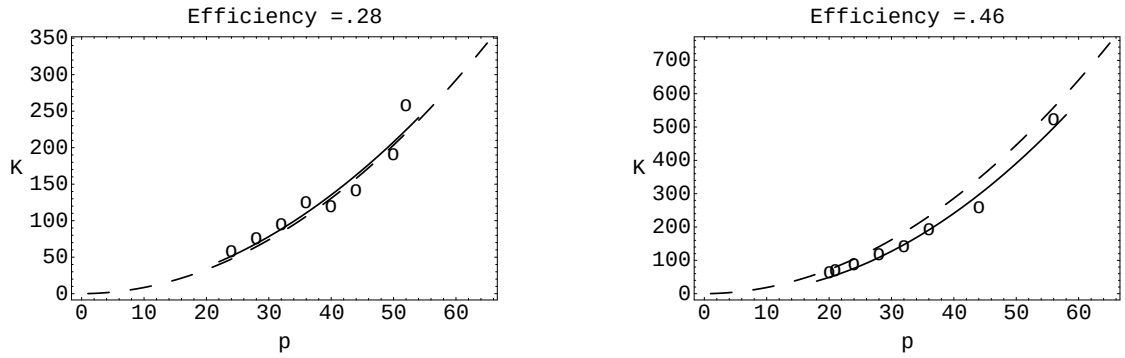

Figure 5. Isoefficiency curves for HG_P (k = 35). Dashed line — theoretical; solid line — fit to data.

For the Intel Paragon, the linear model is $T_1 = 0.33W'$ with the standard deviation of errors equal to 2.25. The regression explains 99% of the variation in T_1. The standard deviation of values T_{comm} observed in each repetition of the experiment was very large, which is characteristic to the Intel Paragon design. Nevertheless, if the message length is known or lies within a certain range, some constant value of T_{comm} can be estimated by the linear regression within that specific range. In particular, if a small array of double precision constants is transmitted, the communication latency can be approximated by the start-up time, i.e., $T_{comm} = 605\mu$s.

For the Cray T3E, the linear model is $T_1 = 0.019W'$ with the standard deviation of errors equal to 0.07. The regression explains 99% of the variation in T_1. To estimate the communication time T_{comm}, the linear regression $T_{comm} = 50 + 0.16L$ is obtained. For this regression, the standard deviation of errors is 6.3 and the coefficient of determination is 0.91.

As a result, the times spent for computing a million floating point operations are 0.024s, 0.33s, and 0.019s for the IBM SP2, Intel Paragon, and Cray T3E, respectively. The reciprocal of t_c defines uniprocessor computing rates of 41.6 Mflop/s, 3.0 Mflop/s, and 53 Mflop/s, correspondingly. Start-up–transmission time pairs (in microseconds) are (69, 0.45) and (50, 0.16) for the IBM SP2 and Cray T3E, respectively.

The experimental results are presented as a series of graphs, Figures 4–9, where the dashed line indicates the relationship between K (the scaling factor) and p (the number of processors) for a fixed value of the efficiency E. The solid line is a least squares fit of the experimental data. In all cases the vertical axis represents K and the horizontal axis p.

4.2. Isoefficiency on the Intel Paragon

Least squares fits to HG_P data in Figure 4 (top) for efficiencies .28, and .46 with $k = 15$, respectively, are $0.03p^2 + 1.98p$ and $0.13p^2 + 0.05p$. Least squares fits to MHG_P data in Figure 4 (bottom) for efficiencies .28, and .46 with $k = 15$, respectively, are $0.05p^2 + 1.58p$ and $0.13p^2 + 0.54p$. Predicted isoefficiency functions for the same efficiences and value of k, respectively, are $0.07p^2 + 0.04p$ and $0.15p^2 + 0.08p$.

Least squares fits to HG_P data in Figure 5 for efficiencies .28, and .46 with $k = 35$, respectively, are $0.08p^2 + 0.28p$ and $0.18p^2 - 1.14p$. Predicted isoefficiency functions for the same efficiences and $k = 35$, respectively, are $0.08p^2 + 0.04p$ and $0.18p^2 + 0.09p$. For the Intel Paragon parallel systems, each least squares approximation grows similarly to the corresponding predicted isoefficiency. Thus, the same function can be used to estimate the isoefficiency scalabilities of MHG_P and HG_P. As

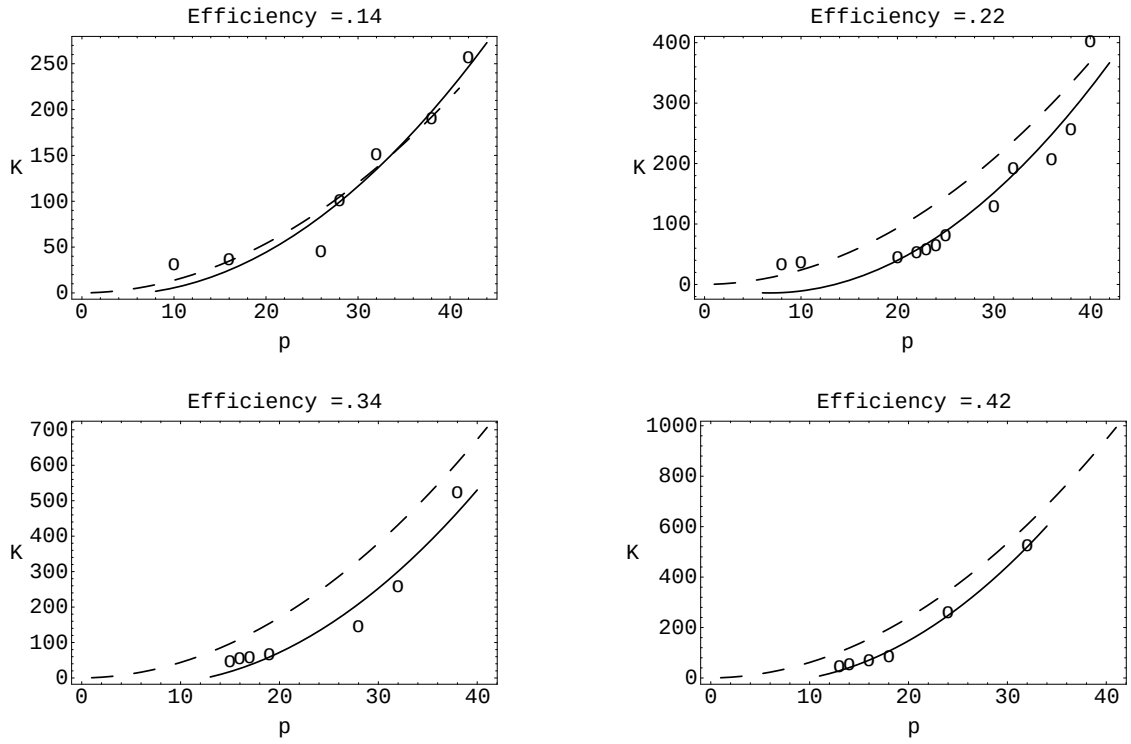

Figure 6. Isoefficiency curves for HG_S (k = 15). Dashed line — theoretical; solid line — fit to data.

reflected in the form of the isoefficiency function, the larger the efficiency to be maintained, the larger the increase in the problem size required with scaling of an architecture component.

When the restart parameter k increases, the leading term coefficient also increases. This variation in the isoefficiency function value is in agreement with consideration of the whole restart cycle in the isoefficiency analysis of GMRES(k), which differs in this sense from the analysis of the preconditioned conjugate gradient method conducted in [8] for a single iteration of the method.

4.3. Isoefficiency on the IBM SP2

Least squares fits $f_E(p)$ of a quadratic polynomial to HG_S data in Figure 6 for efficiencies .14, .22, .34, and .42 with $k = 15$, respectively, are $0.17p^2 - 1.11p$, $0.31p^2 - 4.15p$, $0.48p^2 - 6.06p$, and $0.74p^2 - 7.42p$. Least squares fits to MHG_S Figure 7 data for efficiencies .14, .22, .34, and .42 with $k = 15$, respectively, are $0.10p^2 + 5.15p$, $0.17p^2 + 3.14p$, $0.40p^2 - 1.17p$, and $0.56p^2 - 4.31p$. Predicted isoefficiency functions for the same efficiences and the same value of k, respectively, are $0.13p^2 + 0.07p$, $0.23p^2 + 0.13p$, $0.42p^2 + 0.23p$, $0.58p^2 + 0.33p$.

Least squares fits to MHG_S data in Figure 8 for efficiencies .22 and .42 and $k = 25$, respectively, are $0.24p^2 + 2.03p$ and $0.75p^2 - 5.06p$. Predicted isoefficiency functions for efficiences .22 and .42 with $k = 25$, respectively, are $0.26p^2 + 0.14p$ and $0.66p^2 + 0.36p$. Each least squares approximation grows in accordance with the corresponding predicted isoefficiency.

4.4. Isoefficiency on the Cray T3E

Least squares fits to HG_T data in Figure 9 (top) for efficiencies .24 and .36, respectively, are $0.45p^2 - 1.87p$ and $0.70p^2 - 5.36p$. Least squares fits to MHG_T data in Figure 9 (bottom) for

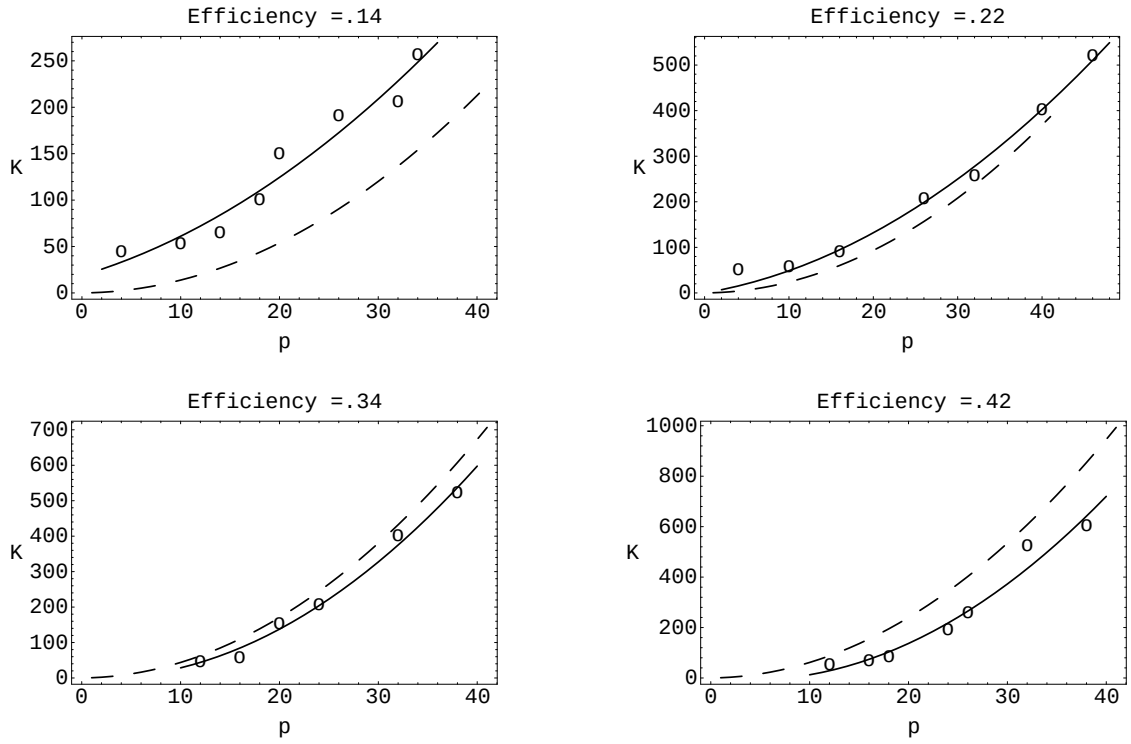

Figure 7. Isoefficiency curves for MHG_S (k = 15). Dashed line — theoretical; solid line — fit to data.

efficiencies .24 and .36, respectively, are $0.32p^2 + 1.87p$ and $0.58p^2 - 2.71p$. Predicted isoefficiency functions for the same efficiences, respectively, are $0.33p^2 + 0.18p$ and $0.59p^2 + 0.32p$. For the Cray T3E parallel systems, each least squares fit grows similarly to the corresponding predicted isoefficiency. Since the architectures considered here have different values of machine-dependent constants, the corresponding isoefficiency functions have different coefficients, even though they include terms of the same order. For example, the isoefficiency function f_{E_T} on the Cray T3E differs significantly from the isoefficiency function f_{E_S} on the IBM SP2. In particular, for $E = 0.42$, $f_{E_T} = 0.76p^2 + 0.48p$ and $f_{E_S} = 0.58p^2 + 0.33p$. The coefficient of the leading term of f_{E_T} is larger than that of f_{E_S} since the computing rate $1/t_c$ on the Cray T3E is faster than on the IBM SP2, while the improvement in t_s and t_w is not sufficient to hide the communication latency. Hence, the idling time on the Cray T3E is larger than on the IBM SP2. The isoefficiency function of the Paragon architecture presents the case when moderate values for *all* the machine-dependent constants involved result in good isoefficiency scalability characteristics. However, with scaling of the problem size, less powerful architectures, such as the Intel Paragon, tend to exhaust their resources faster than more powerful ones, such as the Cray T3E and IBM SP2.

4.5. Isoefficiency for adaptive GMRES(k)

Sections 4.2–4.4 and Figures 4–9 pertained to parallel restarted GMRES(k), where k did not adapt during the linear system solution process. Results for parallel adaptive GMRES(k) are summarized in this section; figures for parallel adaptive GMRES(k) analogous to Figures 4–9 are in [16] and [5]. The scatter of the (p, K) data for parallel adaptive GMRES(k) is such that no meaningful least squares fit (polynomial or otherwise) is possible. The details of the scalability

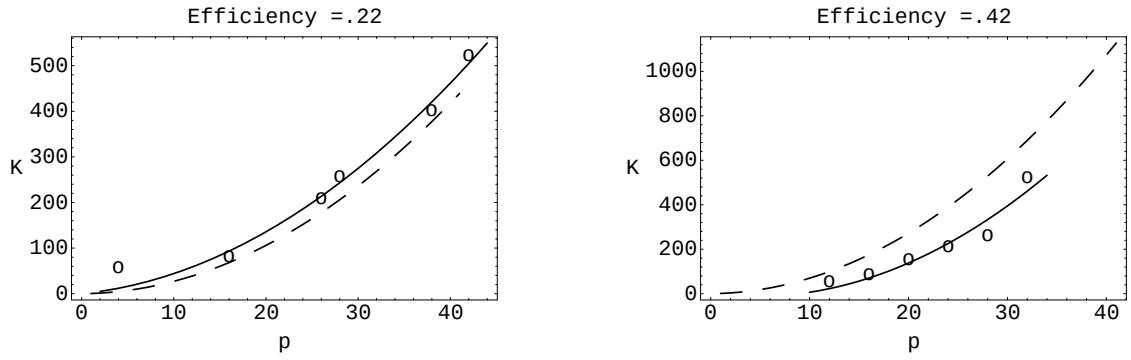

Figure 8. Isoefficiency curves for MHG_S (k = 25). Dashed line — theoretical; solid line — fit to data.

analysis for parallel adaptive GMRES(k) are similar to the derivation in §3, and can be found in [16]; a brief discussion follows.

In the case of problem size scaling considered here, changing matrix dimensions (K times) can lead to a variation in test problem difficulty as well as in its size. This often happens in practical applications, such as postbuckling analysis of structures, with increase in problem size. The erratic behavior of the efficiency values for adaptive GMRES(k) on the Paragon suggests that not only the convergence for adaptive GMRES(k) differs from that for the restarted GMRES (see Section 2), but also their parallel algorithm-architecture performances differ.

By definition, adaptive GMRES(k) proceeds qualitatively differently on different problems. Therefore, for adaptive GMRES(k), the operation count of the full convergence history (or of the convergence history until the maximum subspace dimension is reached) represents the useful work to be considered in the isoefficiency analysis. Otherwise, since it is not known in advance when increases in k occur during the solution process, the isoefficiency characteristics of adaptive GMRES(k) are unpredictable [16]. Variation in the Krylov subspace dimension makes the scalability analysis of the adaptive GMRES(k) algorithm more complicated than the analysis of the conjugate gradient method [8], where an operation count and parallel overhead per iteration are sufficient to derive an isoefficiency function. Likewise, the definition of both useful work and parallel overhead differ from their expressions in the analysis of GMRES(k), which can be performed for a particular subspace dimension k and a particular operation count of a GMRES iteration.

One approach to estimating the efficiency of the adaptive GMRES(k) algorithm on scaled problems is to introduce a measure of problem difficulty into the problem size definition. Since it is hard to predict convergence of the adaptive GMRES(k) method on an arbitrary problem, making problems more uniform by preconditioning is another way of dealing with the issue of useful work scaling for the purpose of an isoefficiency analysis.

5. Conclusions

In this paper, an isoefficiency analysis is carried out for parallel versions of GMRES(k) with Householder reflection orthogonalization implemented on the Intel Paragon, IBM SP2, and Cray T3E. The theoretical part of this analysis not only establishes an asymptotic relation between the increase in problem size and number of processors, but also provides an analytic expression for the isoefficiency function. Communication and nonessential work overheads are identified for parallel GMRES(k) applied to solve a real-world circuit simulation problem distributed among

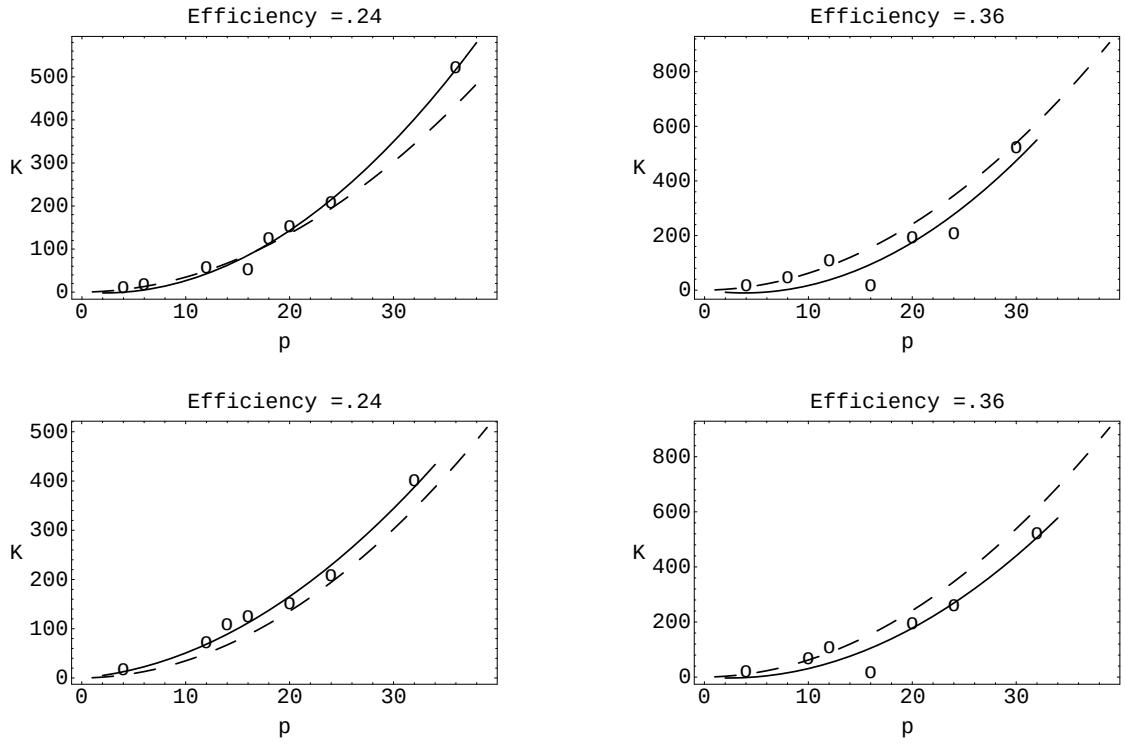

Figure 9. Isoefficiency curves for HG_T (k = 15, top) and MHG_T (k = 15, bottom). Dashed line — theoretical; solid line — fit to data.

processors in a block-striped fashion. For vertex-based partitioning of the problem, the theoretical overhead is claimed to be the same under certain assumptions on load balancing by a partitioning algorithm. Thus, the same isoefficiency function (same structure, different coefficients) is derived for the parallel GMRES(k) implementations used with each of these two partitioning schemes. On target parallel architectures, experimental results support the claim and *closely match predicted isoefficiency functions.*

Both theoretical and experimental results show that the isoefficiency function is a quadratic polynomial with a small leading term coefficient, which implies that the given parallel GMRES(k) implementations are reasonably scalable on the given parallel architectures, namely, on an IBM SP2, Intel Paragon, and Cray T3E. In general, a parallel algorithm is considered scalable if the isoefficiency function (in any form) exists; that is, given a parallel architecture and a fixed efficiency value, the amount of useful work in a given algorithm can be determined such that a given parallel system achieves the desired efficiency.

The results here strongly support the validity of isoefficiency as a tool for scalability analysis of parallel restarted GMRES, but not for parallel adaptive GMRES(k). A battery of tests in [16], corresponding to those represented here by Figures 4–9, show that isoefficiency functions can not be computed for adaptive algorithms applied to problems whose difficulty varies. In particular, for parallel adaptive GMRES(k), the results in [16] indicate a strong dependency of the efficiency of a parallel version of the method on the variability in problem difficulty when problems are scaled. Thus (1) a measure of the problem difficulty has to be a parameter of a scaling procedure, or (2) scalable preconditioning has to be applied to normalize the problem difficulty with increasing problem size, or (3) isoefficiency is not a meaningful concept for adaptive algorithms.

A final comment about empirical isoefficiency analysis is that because of resource saturation with useful work scaling up, there is an upper bound on the problem size beyond which the sequential execution time is affected by memory hierarchy operations. In this case, the time measurement of solving large problems on a uniprocessor may impede the isoefficiency analysis. The efficiency becomes greater than one and the relation (1) between the useful work and the parallel overhead cannot be applied to study scalability of an algorithm-machine combination (this does not apply to the data in this paper).

References

[1] Å. Björck. Solving linear least squares problems by Gram-Schmidt orthogonalization. *BIT*, 7:1–21, 1967.

[2] C. H. Bischof and P. T. P. Tang. Robust incremental condition estimation. *Tech. Rep. CS-91-133, LAPACK Working Note 33*, Computer Sci. Dept., Univ. of Tennessee, May, 1991.

[3] P. N. Brown and H. F. Walker. GMRES on (nearly) singular systems. *SIAM J. Matrix Anal. Appl.*, 18(1):37–51, January 1997.

[4] J. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. *Math. Comp.*, 30:772–795, 1976.

[5] M. S. Driver, D. C. S. Allison, and L. T. Watson. Scalability of adaptive GMRES algorithm. in *Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing*, CD-ROM, SIAM, Philadelphia, PA, 1997, 7 pages.

[6] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore MD, 2nd ed., 1989.

[7] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distrib. Technol.*, 1:12–21, 1993.

[8] A. Gupta, and V. Kumar. Performance and scalability of preconditioned conjugate methods on parallel computers. *IEEE Trans. Parallel and Distrib. Systems*, 6:455–469, 1995.

[9] G. Karypis and V. Kumar. MeTiS: Unstructured graph partitioning and sparse matrix ordering system. *User's Guide—Version 2.0*, Dept. of Computer Sci., Univ. of Minnesota, Minneapolis MN 55455, 1995.

[10] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide*. SIAM, Philadelphia, PA, 1998.

[11] G.-C. Lo and Y. Saad. Iterative solution of general sparse linear systems on clusters of workstations. *Tech. Report, UMSI 96/117*, Supercomputer Institute, Univ. of Minnesota, 1200 S. Washington Ave., Minneapolis MN 55415, August 1996.

[12] Y. Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, 1996.

[13] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869, 1986.

[14] R. B. Sidje. Alternatives for parallel Krylov subspace basis computation. *Numer. Linear Algebra Appl.*, 4:305–331, 1997.

[15] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: methodology and examples. *Computer*, 7:42–50, 1993.

[16] M. Sosonkina. Parallel sparse linear algebra for homotopy methods. *Ph.D. Thesis*, Computer Sci. Dept., Virginia Tech, Blacksburg VA 24061, September 1997.

[17] M. Sosonkina, L. T. Watson, R. K. Kapania, and H. F. Walker. A new adaptive GMRES algorithm for achieving high accuracy. *Numer. Linear Algebra Appl.*, 5:275–297, 1998.

[18] H. A. van der Vorst and C. Vuik. The superlinear convergence behaviour of GMRES. *J. Comp. Appl. Math.*, 48:327–341, 1993.

[19] H. F. Walker. Implementation of the GMRES method using Householder Transformations. *SIAM J. Sci. Stat. Comput.*, 9:152–163, 1988.