A High Performance Parallelization Scheme for the Hessenberg Double Shift QR Algorithm

Reiji Suda[a], Akira Nishida[b] and Yoshio Oyanagi[b]

[a]*Department of Computational Science and Engineering, Graduate School of Engineering, Nagoya University, 464-8603, Japan*

[b]*Department of Information Science, the University of Tokyo, 113-0033, Japan*

Abstract

We propose a new parallelization scheme for the Hessenberg double shift QR algorithm. Our scheme allows software pipelining and communication latency hiding, and gives almost perfect load balance. An asymptotic parallelizing overhead analysis shows that our scheme attains the best possible scalability of the double shift QR algorithm, and that the overheads are less than the multishift algorithm when $n = \omega(p^2)$, where n is the matrix size and p is the number of processors. Its high exploitation of the parallelism of the double shift QR algorithm is demonstrated by an implementation on Fujitsu AP1000+ multicomputer system.

Key words: Double shift QR algorithm; parallel processing; data mapping; performance analysis.

1 Introduction

The most reliable eigensolver for nonsymmetric real matrices is the combination of the Hessenberg reduction by Householder transformations and the double shift QR algorithm. Many matrix computation algorithms allow efficient parallelization with regular block/cyclic data mapping. The Hessenberg reduction by Householder transformations can be efficiently parallelized with those usual schemes. Using the same strategy, however, one can obtain no significant parallel speedup for the Hessenberg double shift QR algorithm.

Fig. 1 depicts some data mapping schemes proposed for the Hessenberg double shift QR algorithm. Van de Geijn [8] proposed the block Hanckel-wrapped

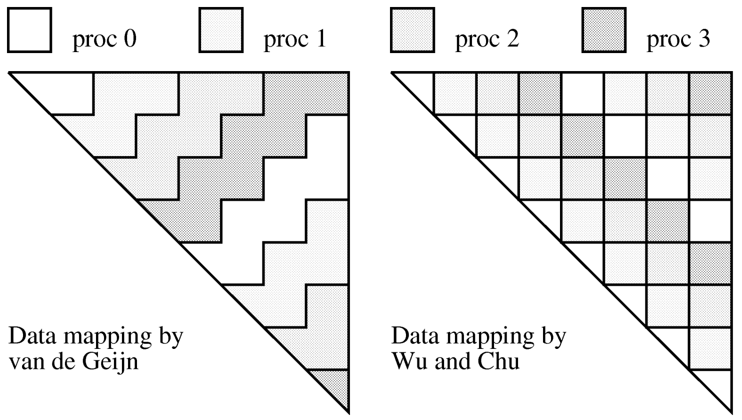

Fig. 1. Data mapping schemes in some researches ($p = 4$).

storage scheme, in which the (i, j) block is allocated to the processor $(i + j)/m \bmod p$, where m is a positive integer, and p is the number of processors. The left figure of Fig. 1 shows his data mapping with $p = 4$ and $m = 2$. Wu and Chu [9] allocated the blocks along the main diagonal, as shown in the right figure of Fig. 1. In their mapping, the load balance and the communication overheads are a little worse than the block Hanckel-wrapped storage scheme.

Henry and van de Geijn [4] reported an implementation of parallel Hessenberg double shift QR algorithm with the block Hanckel-wrapped storage scheme. The performance was much better than the preceding research results [1–3,9], but the parallel efficiency was not satisfactory. The major overhead that determines the parallel efficiency of their implementation was the idle time waiting for the computations of the *look-ahead steps* (computing transformations and rotating rows on diagonal blocks) and the broadcasts of the results of those steps. A pair of well-known techniques can eliminate that idle time — software pipelining and communication latency hiding. However, the block Hanckel-wrapped storage scheme is incompatible with those techniques. In order to attain higher parallel efficiency, we propose a new data mapping, which is an improvement of the block Hanckel-wrapped storage scheme and allows software pipelining and communication latency hiding.

Our parallelization scheme will be introduced in the next section. Section 3 analyzes its parallelization overheads, and compares it with the multishift QR algorithm. In section 4, we discuss an implementation of our scheme on a multicomputer system. High parallel efficiency of our scheme will be demonstrated with experimental results. In this paper, n and p denotes the matrix size and the number of processors, respectively.

Another approach to higher parallel efficiency for the QR algorithm is increasing the parallelism with *multiple shifts* [5]. However, the parallelization of the double shift QR algorithm is still important in three points. First, the multishift algorithm converges differently with a different number of shifts. The

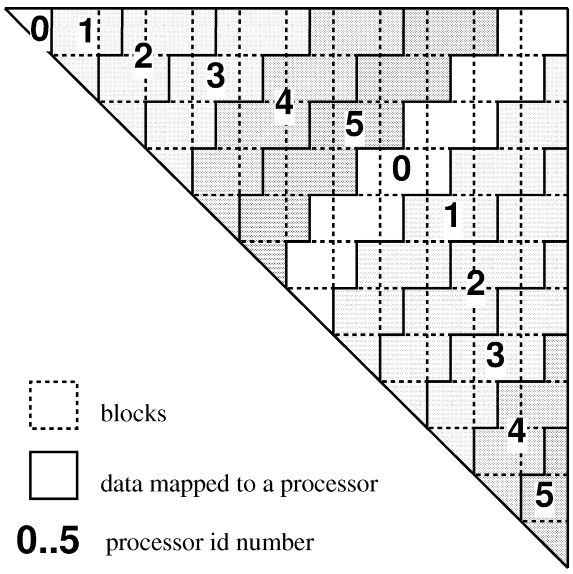

Fig. 2. The proposed data mapping ($p = 6$).

speedup of the parallel multishift QR algorithm is difficult to predict, since the best number of shifts must depend on the number of processors. A parallel double shift QR algorithm will provide *more reliable* speedup in this sense. Second, the parallel double shift algorithm will be faster than the parallel multishift algorithm on certain coarse grain conditions, as will be discussed in this paper. Third, some techniques developed for the double shift algorithm may be applicable to the multishift algorithm, possibly with extension. For those reasons, we worked on parallel processing of the *double shift* QR algorithm.

2 A new parallelization scheme for the Hessenberg double shift QR algorithm

In this section, our parallelization scheme for the Hessenberg double shift QR algorithm [7,6] is explained. The scheme consists of three parts: the mapping of the matrix elements to the processors, the allocation of the tasks to the processors, and the scheduling of the tasks. It will be shown that our scheme allows good load balance, software pipelining, communication latency hiding, and highly efficient parallel processing.

2.1 Data mapping

Fig. 2 depicts the proposed data mapping for $p = 6$. It is based on the partition of the matrix into $2p \times 2p$ *blocks*, that are shown with the dotted lines. The solid lines show the boundaries of the matrix regions allocated to different

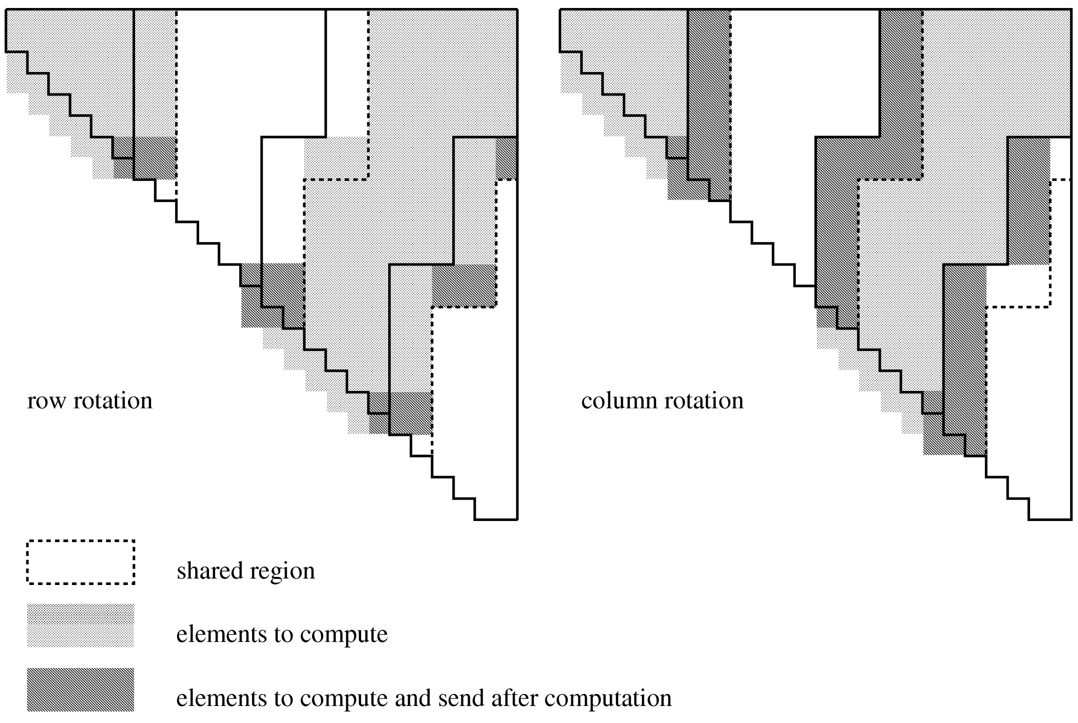

Fig. 3. The task allocation ($p = 2$, $n = 24$).

processors, and the numbers **0**-**5** indicate to which processor each region should be allocated. The mapping is similar to the block Hanckel-wrapped storage scheme in that the matrix is partitioned into $2p$ *strips* along the subdiagonal, and that each processor owns two strips in cyclic fashion. However, the strips are shifted to left by 1.5 blocks, and this shift makes the loads near the diagonal so light that the look-ahead steps can be executed in advance.

2.2 Task allocation

We use the *owner-computes rule* for the task allocation as depicted in Fig. 3, where $n = 24$ and $p = 2$. The solid lines show the boundaries of the data mapping. The processor 0 updates the gray elements, and sends the dark gray elements to the processor 1 after the computation. In order to minimize the total message size, the task allocations for row/column rotations are slightly different. The broken lines depict the region shared with the neighboring processors. The width of the shared region is two elements.

2.3 Task scheduling

Fig. 4 depicts the task scheduling. The rotations on the gray elements are bundled and form a *block transformation*. Because the boundaries of the data

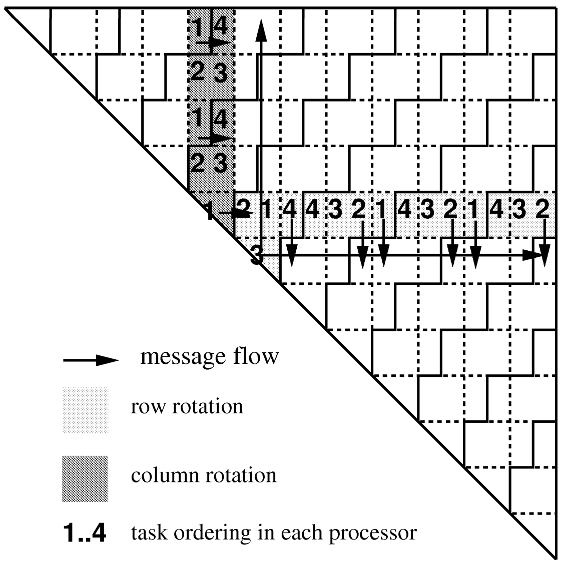

Fig. 4. The task scheduling for the sixth block transformation ($p = 6$).

mapping run the middle of blocks, it is convenient to consider the rotations on a *half block* as a unit task. We call the time for that unit task *quarter*, because each processor computes rotations on four half blocks in a block transformation.

Fig. 4 shows the scheduling of the tasks in the sixth block transformation. Each processor has four half blocks for rotations, and the order of the rotations is shown with the numbers **1** to **4**. The scheduling is local, and no synchronization among the processors is taken except the message flow. The arrows depict some of the message flow sent after computations (not all, for simplicity). The long arrows from the diagonal block stand for the broadcast of the results of the look-ahead step.

The look-ahead step is of the next block transformation. That is software pipelining, and removes the idling time of the other processors waiting for the look-ahead step computations. It is executed in the third quarter, so that there is a quarter between the end of the look-ahead step and the beginning of the rotations that use the results (because results of the computation in the **3**rd quarter is used in the **1**st quarter of the next block transformation: We express that by $3 \rightarrow 1^+$). Therefore, it is possible to hide the latency of the broadcasts in a quarter of a block transformation.

The row rotations in a processor are executed from right to left, because the results of the computations on the right two half blocks must be sent to the next processor. With this ordering, four quarters ($2 \rightarrow 3^+$) are available to hide the communication latency. The marginal time becomes shorter in some blocks near the diagonal, but a quarter ($4 \rightarrow 2^+$) still remains. The column rotations in a processor are executed from bottom to top, because the next

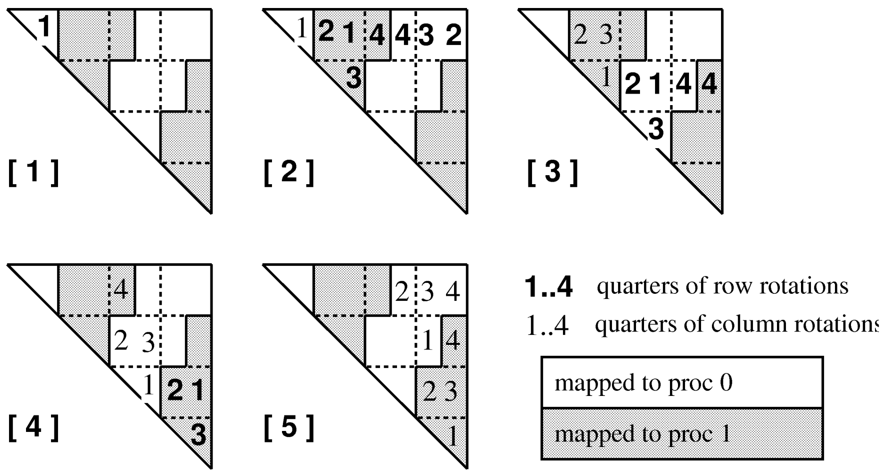
[1]
[2]
[3]
[4]
[5]
1..4 quarters of row rotations
1..4 quarters of column rotations
mapped to proc 0
mapped to proc 1

Fig. 5. The whole story of the task scheduling ($p = 2$).

processor uses results of the computations of the half block at the bottom in the fourth quarter of the same block transformation. Therefore, the latency of the communication can be hidden with two quarters ($1 \rightarrow 4$).

The *backward* communication (from a processor to the previous processor: not shown in the figure) have longer marginal time than the *forward* communication (from a processor to the next processor: short arrows in the figure) and the broadcasts. Thus we see that there is at least a quarter to hide the latency of each communication.

The whole story of the task scheduling for $p = 2$ is shown in Fig. 5, where two exceptions of the explained scheduling are shown: The first exception is at the first block transformation, the processor 0 computes the look-ahead step, and the other processors wait for the message from the processor 0. The second exception is at the second last block transformation, the column rotations of the processor $p-1$ is scheduled at the fourth quarter, and column rotations of a half block on the processor 0 must be postponed to the last block transformation. Those exceptions cause some load imbalance.

For large p, the load imbalance becomes less significant because it is $O(1/p)$ of the parallel execution time. Even for small p, the load imbalance can be alleviated by subdividing the strips. Subdivision also increases the marginal time for the communication latency hiding. An asymptotic analysis on the effects of subdivision will be discussed in the next section.

Fig. 6 shows a trace of our implementation (discussed in section 4) for a Francis step with $n = 800$ and $p = 4$. The communication latency was successfully hidden, and no long idle time waiting for a message was found in the trace except at the first and the last two block transformations. From the fact that the computations of the processors are finished almost at the same time without any synchronization other than the message flow, one can see that

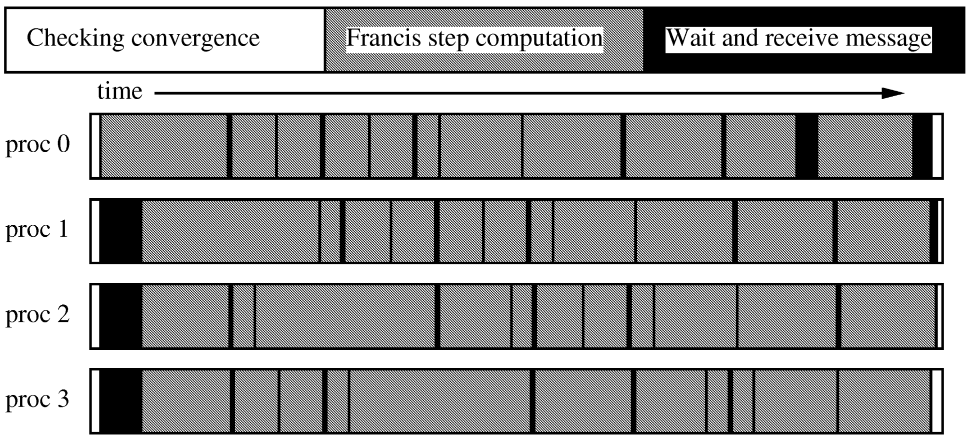

Fig. 6. A trace of the computations and the communications in a parallel Francis step ($p = 4$, $n = 800$; The time for sending messages is included in the computation time).

the loads of the processors are nicely balanced.

3 An analysis on the parallelizing overheads of parallel QR algorithms

In this section, we analyze the overheads of our scheme, and compare it with the parallel multishift algorithm [5]. We define the *parallelizing overheads* $\mathcal{O}$ as the time for idling and extra tasks that are not necessary for serial processing. It is expected that $\mathcal{O} = T_p - T_1/p$, where T_p is the parallel processing time with p processors, and T_1 is that for $p = 1$.

In the following analyses, we assume infinitely large n and p, and thus ignore the lower order terms and the constant factors. (As for the reason, see section 3.4.) Some parameters will be optimized before the comparison.

First we review the asymptotic notations. Readers may be familiar with O and o notations: $f(n) = O(g(n))$ means $f(n) \leq cg(n)$ and $f(n) = o(g(n))$ means $f(n) < cg(n)$, both for some constant c and for large enough n. Ω and ω are inverse relations of O and o, respectively: $f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$, $f(n) = \omega(g(n)) \iff g(n) = o(f(n))$ The exact order is represented by Θ: $f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Roughly speaking, o, O, Θ, Ω, and ω are similar to $<$, $\leq$, $=$, $\geq$, and $>$, respectively.

3.1 The parallelizing overheads of the double shift QR algorithm

In our scheme, five kinds of overheads are significant in large problems. Let the *block size* be $H = n/(2hp)$ and the *subblock size* be $B = n/(2bp)$: The matrix is split into $2bp$ strips with width B, and they are mapped to the processors in block-cyclic fashion. Consecutive b/h strips are mapped to a processor, so that the width of each block strip is H. Note that $1 \leq h \leq b \leq n/(2p)$.

Load imbalance. The major load imbalance of the parallel double shift QR algorithm is at the first look-ahead step, which takes $\Theta(B^2) = \Theta(n^2/(b^2p^2))$ time.

Broadcasts. The results of the look-ahead steps must be broadcasted. The transfer time becomes larger than the startup time for large problems, and the overhead time is $\Theta(n)$, assuming the circular broadcast method (which will be discussed in section 4).

Data exchanges. The data in the shared region are exchanged after the computation, as is shown in Fig. 3. The total amount of the exchanged data at a processor is $\Theta(hn)$, which is the area of the shared region of a processor. However, this overhead is negligible when the communication latency is successfully hidden.

Short loop effects. The core routine of the QR algorithm is a double loop, and its performance is much affected by the inner loop length. Non-floating point operations and CPU pipeline stall reduce the flops performance when the inner loops are short. This kind of overhead can be evaluated as $\Theta(hn)$, which is the total number of the outer loop iterations.

Data redistribution. Since the proposed data mapping is quite unique, data redistribution will be required before executing the Francis steps. The worst-case time consumption for the data redistribution is $\Theta(n^2)$, which is the number of matrix elements. Because usually the double shift QR algorithm needs $\Theta(n)$ Francis iterations, the *amortized* overhead is expected to be $\Theta(n)$ per Francis iteration.

Summing up the above, the overheads per Francis iteration amount to $\Theta(n^2/(bp)^2 + hn)$. This is minimized to $\Theta(n)$ by letting $h = \Theta(1)$ and $b = \Theta(n/p)$. With this parameter choice, the subblock size is constant, and the subdivided strips are allocated in block fashion. Since the overhead is $\Theta(n)$ and the load per processor is $\Theta(n^2/p)$, constant parallel efficiency is obtained with $p = \Theta(n)$. Therefore, our scheme attains the best possible scalability of the double shift QR algorithm: The number of processors that gives constant parallel efficiency is the same order as the parallelism of the algorithm.

3.2 The parallelizing overheads of the multishift QR algorithm

Next, the parallelizing overheads of the multishift QR algorithm are evaluated. The proposition paper [5] has already discussed it, but the short loop effects, the shift computations, and the parameter optimizations were not included. We will consider them, but still ignore the difference of the convergence. We also assume that the processors are arranged in a $\sqrt{p} \times \sqrt{p}$ square array. Let the number of shifts $S = \sigma\sqrt{p}$, the block size $H = n/(h\sqrt{p})$, and the subblock size $B = n/(b\sqrt{p})$. (Note that the parameters are not the same as those of the previous analysis on the double shift algorithm, though closely related.) Again note that $1 \leq \sigma \leq n/\sqrt{p}$ and $1 \leq h \leq b \leq n/\sqrt{p}$. Under those assumptions, the major overheads can be evaluated as follows:

Look-ahead steps. The off-diagonal processors idly wait while the diagonal processors compute the look-ahead steps. The overheads are $\Theta(B^2 b) = \Theta(n^2/(bp))$, amortized per shift.

Pipeline startup time. The pipeline startup and wind-down overheads are evaluated in [5] as $\Theta(Hn/S) = \Theta(n^2/(\sigma ph))$ per shift.

Broadcasts. The time for broadcasts is $\Theta(n \log p/\sqrt{p})$ per shift, as evaluated in [5].

Data exchanges. The costs for the exchanges of the data in the border region are $\Theta(hn/\sqrt{p})$. The **short loop effects** gives the same order of overheads.

Shift computations. We assume that the double shift QR algorithm is used for the shift computation (to avoid recursion that makes the analysis complex and unrealistic), and that the matrix elements are not redistributed. Further, let us assume $S \leq H\sqrt{p}$; that is, some processors are idle because the size of the submatrix is too small. That inequation is satisfied with the optimized parameters shown below. On those assumptions, the major computational costs of the shift computation are from the rotations, which are $\Theta(S^2H)$. Therefore, the overheads per shift are $\Theta(SH) = \Theta(\sigma n/h)$.

Thus, the sum of the parallelizing overheads of the multishift algorithm is $\Theta(n^2/(bp) + n^2/(\sigma hp) + n \log p/\sqrt{p} + hn/\sqrt{p} + \sigma n/h)$. To minimize that, b should be the largest possible value, that is $\Theta(n/\sqrt{p})$. With $\sigma = \Theta(\sqrt{n/p})$ and $h = \Theta(n^{1/4})$, the overhead is minimized to $\Theta(n^{5/4}/p^{1/2})$. However, this is valid only for $n = \Omega(p)$, otherwise the condition $\sigma \geq 1$, which means that there are enough parallelism for p processors to work, is not satisfied. Therefore, for $n = o(p)$, σ should be constant. Then the total overhead becomes $\Theta(n/p^{1/4})$ with $h = \Theta(p^{1/4})$.

3.3 Comparison of the parallelizing overheads of the two algorithms

We have seen that the parallelizing overhead of the double shift QR algorithm is $\Theta(n)$, and that of the multishift algorithm is $\Theta(n^{5/4}/p^{1/2})$ for $n = \Omega(p)$ and $\Theta(n/p^{1/4})$ for $n = o(p)$, with the optimum parameter values for each. Therefore, the multishift algorithm is faster than the double shift algorithm for $n = o(p)$. This is a natural result, because the number of processors is larger than the parallelism of the double shift algorithm.

For larger n, the double shift algorithm can be faster than the multishift algorithm. The two methods have the same order of overheads when $n = \Theta(p^2)$. Therefore, the double shift algorithm will be faster for $n = \omega(p^2)$, while the multishift algorithm will be faster for $n = o(p^2)$. Thus, we have proved that the double shift algorithm is advantageous on the coarse grain condition $n = \omega(p^2)$.

3.4 More detailed analysis

It might be possible to make a more detailed analysis as some other papers do [4,5]. However, we found that the parallel execution time of our implementation discussed in the next section does hardly agree with the analysis. On coarse grain conditions, the major overheads come from CPU pipeline stall at the ends of the innermost loops, and are difficult to predict with high precision. On fine grain conditions, the communication latency that cannot be wholly hidden gives the major overhead. The critical path becomes influenced by quite small perturbation, such as the difference of the execution performance of the row/column rotations. The two-direction circular broadcast method (explained in the next section) makes the critical path still harder to predict.

4 A high performance implementation of the proposed scheme

In this section, we report our implementation of the double shift QR algorithm on a multicomputer system and its performance. We implemented it on Fujitsu AP1000+, which is a distributed memory parallel processing system with SuperSPARC processors (50 MHz) connected by a torus network. AP1000+ provides communication library routines with DMA, `put` and `put_stride`, with which a processor can send a message with a very small loss of CPU time (a few μs) and with a throughput of 25 MByte/s. Our program computes exactly the same thing as the routine `HQR` of the EISPACK. The inputs are upper Hessenberg matrices with random numbers as the elements, and the observed convergence was quite stable.

4.1 The performance results

The parallel efficiency is by far the most important evaluation metric for the parallel double shift QR algorithm, because the parallel efficiency of the double shift QR algorithm determines the memory requirements per processor [4]. To attain a constant parallel efficiency, one must fix $m = n/p$, thus the amount of data mapped to a processor must be mn. If the memory size per processor is M, n cannot be more than M/m. Because $m = n/p$, the number of the processors is limited as $p \leq M/m^2$. Thus, the parallel efficiency determines how many processors can be utilized.

The parallel efficiency for p processors $e(p)$ is usually defined as $e(p) = P_p/(pP_1)$, where P_p and P_1 are the computing performance (in flops, in our case) with p processors and a single processor for the same problem, respectively. However, the above definition has a drawback that if the problem size is too large to execute on a single processor then the parallel efficiency cannot be defined. Therefore, we use another definition $\tilde{e}(p) = P_p/(pP_{max})$, where P_{max} is the best performance observed on a single processor. We have $\tilde{e}(p) \leq e(p)$ because $P_1 \leq P_{max}$. In our case, although the peak performance of a processor is 50 Mflops, we could observe no performance higher than 20 Mflops for double shift QR algorithm (though tuned with unrolling, tiling, etc.). So we use $P_{max} = 20$ Mflops in the following discussion.

Fig. 7 shows the parallel performance of our program for the first Francis step, where the matrix size reduction (explained in the next subsection) does not occur. The graph shows the relations between the performance P_p/p (y-axis) and the matrix size per processor n/p (x-axis) for several values of p. The parallel efficiency of 50% is attained with $n/p \approx 50$, 80% with $n/p \approx 100$, and 90% with $n/p \approx 150$. Such high parallel efficiency has rarely been

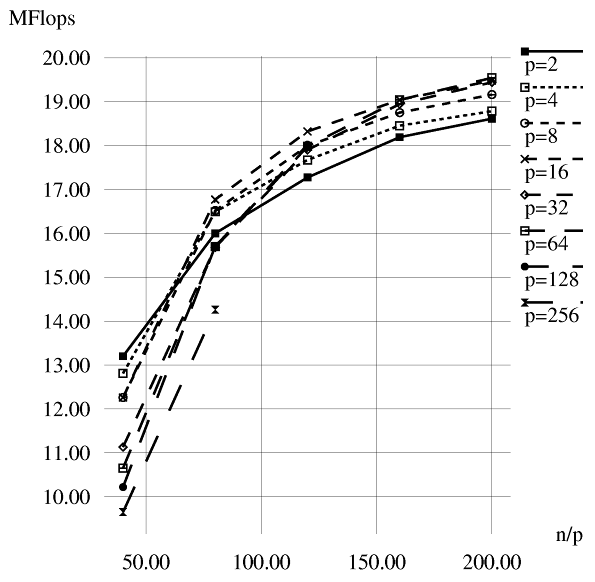

Fig. 7. Performance in Mflops per processor vs. n/p, measured at the first parallel Francis step.

observed in preceding researches on parallel double shift QR algorithm [2,9,4]. The apparent parallel efficiency will become still higher when the eigenvectors are also computed, because the parallelizing overheads for the eigenvector computations are almost negligible.

The performance is mainly determined by n/p, and becomes lower for a smaller n/p. We think that the major overhead for small n/p comes from the short loop effects. The prime cause for the lower performance for a smaller p is the load imbalance, which amounts to $O(1/p)$ of the total computation. The performance for small p may be improved by the subdivision of the matrix partition. The performance seems not decrease with a large number of processors: It agrees with our analysis discussed in section 3.

4.1.1 The broadcast methods

On AP1000+, we can choose the broadcast routine from three alternatives: (1) the hardware-supported broadcast routine, (2) the software broadcast library routine, and (3) a user-defined broadcast routine. The library software routine is not good for our purpose, because it synchronizes the processors before the communication, and thus the communication latency is not hidden. The hardware broadcast of AP1000+ requires no synchronization, but the latency and the throughput are much worse than the point-to-point communication. A software broadcast routine can be written without synchronization, but the processors are weakly synchronized in the relay of the message. In our experiments, the hardware broadcast is faster for large n/p (i.e. large marginal time

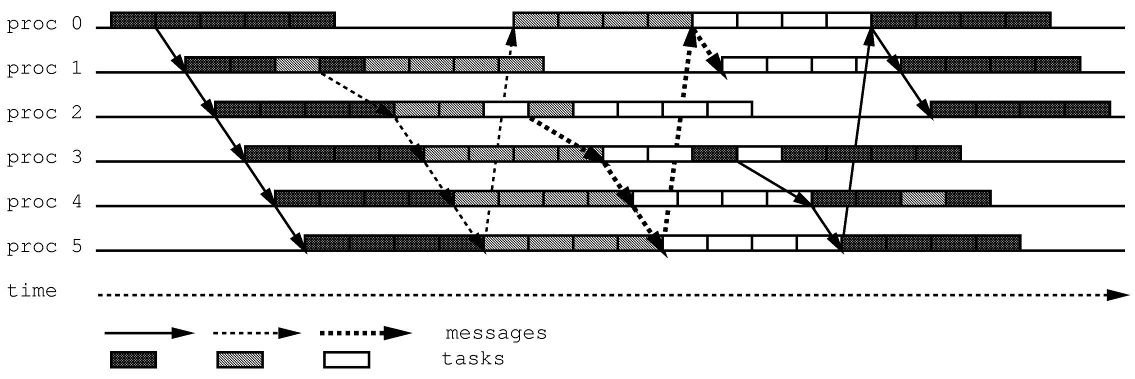

Fig. 8. The circular broadcast method ($p = 6$).

for communication latency hiding), and the user-defined broadcast routine is faster for small n/p.

For user-defined broadcast routine, we used the *circular broadcast method.* Fig. 8 depicts it for $p = 6$. The tasks of each processor are shown on a horizontal line, where a rectangle represents a quarter of a block transformation and an arrow stands for a message for broadcast. The broadcast message is relayed through the processor ring, pipelined with the computations. The relay in Fig. 8 is in one direction, but we use two-direction version in our implementation to halve the latency.

The asymptotic overheads of the circular broadcasts in a Francis step are $\Theta(n)$, which is independent of the number of processors. The time for the first broadcast is $\Theta(n/b)$, because $\Theta(n/(bp))$ data is relayed through $\Theta(p)$ processors. After that, the computations and the communications run in pipeline fashion, and no time for waiting message is spent. The only exception is at the last processor, which is the root processor of the previous broadcast, but the waiting time is $\Theta(n/b)$, and that happens at most $2b$ times in a Francis step for a processor. Therefore, the overhead time for the circular broadcasts is $\Theta(n)$ for a Francis step. The binary-tree broadcast method will require $\Theta(n \log p)$ for a Francis step, and will be slower than the circular method for large p.

4.2 Matrix size reduction in the double shift QR algorithm

Our implementation is a parallelization of `HQR` in the EISPACK. In that routine, the computed region of the matrix is reduced in three ways. The first one is deflation, where the last columns are removed according to the number of the obtained eigenvalues. The dotted arrow **1** and the index **n** in Fig. 9 depict this reduction. Second, the matrix is split into two submatrices at an essentially zero subdiagonal element. The arrow **2** and the index **l** depict the second reduction. Third, the transformations do not begin at the left top cor-

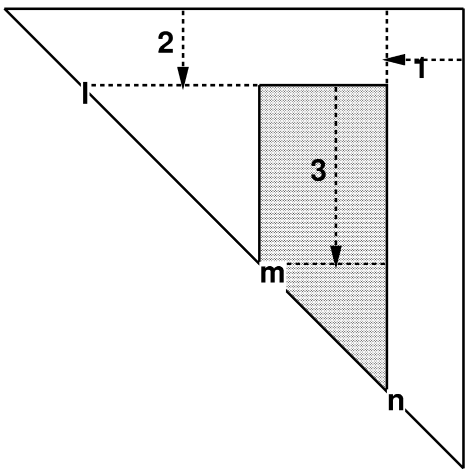

Fig. 9. Matrix size reductions in HQR.

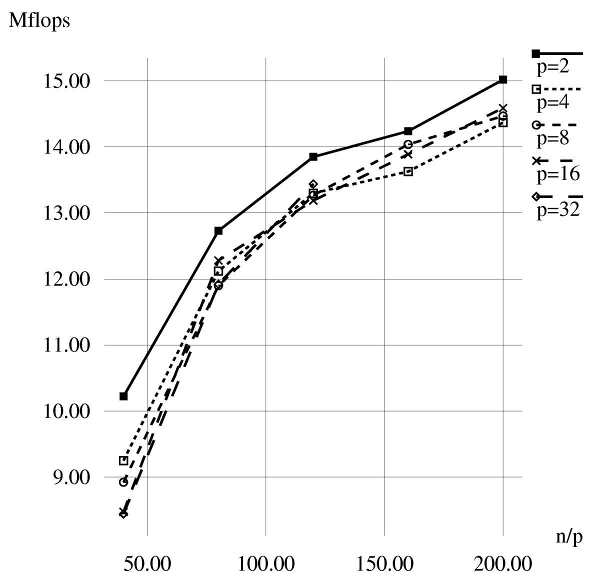

Fig. 10. Performance in Mflops per processor vs. n/p with the block size $n/2p$.

ner when a pair of consecutive small subdiagonal elements exists. The arrow **3** and the index **m** depict this. As a result, the region to be updated becomes the gray trapezoid in Fig. 9.

The third reduction does not influence the load balance of the proposed data mapping, but the others corrupt it. In order to improve the corrupted load balance, we could redistribute the matrix elements. However, we have not yet implemented data redistribution, and attempted to alleviate the performance degradation by subdividing the partition with cyclic mapping.

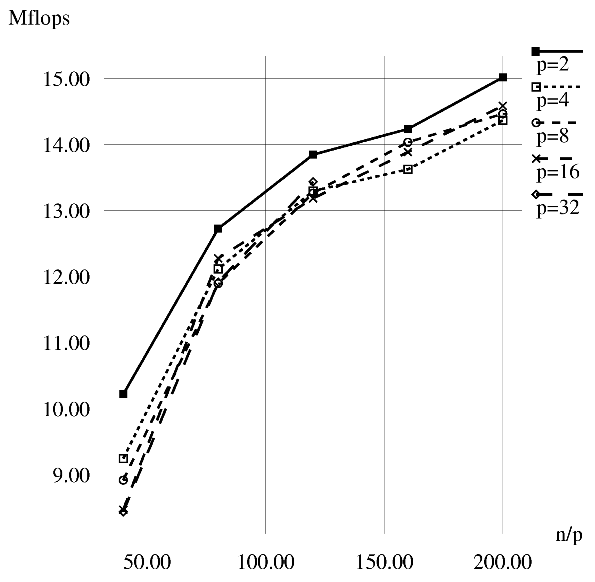

Fig. 11. Performance in Mflops per processor vs. n/p with the best block size.

4.2.1 The performance for the total eigensolution

We ran our program until all the eigenvalues are obtained, and observed the flops performance. First, we set the block size $n/(2p)$. The expected performance is 2/3 of that without matrix size reduction, because the computational complexity is reduced squarely, while the parallel execution time is reduced linearly, with the deflation. The observed performance was better than the expected one, and was 70–80% of the performance without matrix size reduction, as is shown in Fig. 10. We guess that the better performance comes from the marginal time for the communication latency hiding: That may absorb the idle time.

Next, we made a set of experiments with different block sizes, and observed the performance for each block size. The strips are cyclically allocated to the processors. Fig. 11 shows the best performance thus observed. The block size that gives the best performance was about $n/(4p)$ in most cases. The performance is improved more for larger n/p, perhaps because of the less performance degradation with the halved block size. However, the effects of the subdivision become less for larger numbers of processors. We do not have any clear explanation for that, but the $\Theta(\log p)$ overheads for the broadcast is possibly a cause: We used the binary-tree broadcast method because the pipeline of the circular broadcast method is corrupted, and the performance becomes worse than the binary-tree method.

The problems of matrix size reduction do not arise when the full Schur form is computed, as explained in [4]. In that case, the performance of our scheme will be as high as that reported in the previous section. Also, matrix size reduction is less problematic on shared memory machines, because task redistribution would be much easier than on distributed memory machines.

5 Summary and some remarks

We proposed a new parallelization scheme for the double shift QR algorithm. Some of its features are:

(1) communication latency hiding with at least a quarter of a block transformation,
(2) pipelined computations of the look-ahead steps and the rotations,
(3) almost perfect load balance in a Francis step,
(4) best possible scalability for the double shift QR algorithm, and
(5) asymptotically better performance than the multishift algorithm for $n = \omega(p^2)$.

We also demonstrated the efficiency of our scheme by implementing parallel double shift QR program on AP1000+. The attained parallel efficiency was 50% for $n/p \approx 50$, 80% for $n/p \approx 100$, and 90% for $n/p \approx 150$. Although the performance becomes worse for total eigensolution because of the matrix size reduction, such high performance was rarely found with the former parallel implementations of the double shift QR algorithm.

We have discussed parallel processing only on distributed memory parallel processors. However, load balance and software pipelining of the look-ahead steps and the rotations are indispensable for high performance parallel processing on any architecture. Thus, we expect that our scheme will be also necessary for high parallel efficiency on shared memory processors. Our scheme will benefit from shared memory processors: Data redistribution is cost-free, and the task decomposition may be simplified. Marginal time for communication latency hiding will be still useful, serving to absorb some load imbalance. Surprisingly, overhead analysis for shared memory processors differs only in lower order terms and constant factors than that in section 3, and we obtained exactly the same results. Therefore, our scheme will be effective when the number of processors is relatively small, and such is most likely with SMPs.

We have not worked on the multishift QR algorithm, though we are interested in that. The multishift algorithm is efficient on fine grain conditions, but the fact that the double shift algorithm is more advantageous on coarse grain conditions suggests that intermediate algorithms would be better on some conditions, if such algorithms exist.

Acknowledgements

We are grateful to the members of the Oyanagi laboratory, especially to Dr. O. Tatebe, who is now at the ElectroTechnical Laboratory. We sincerely thank the reviewers for their comments. This research was partially supported by Japan Society for the Promotion of Science (JSPS) (Research Promotion Committee CSE, RFTF) and Monbusho Grants-in-Aid for Scientific Research (Nos. 09680327, 09780246, 11780194, 1168031).

References

[1] Z. Bai and J. Demmel, On a block implementation of Hessenberg multishift QR iteration, *International Journal of High Speed Computing*, Vol. 1, No. 1 (1989) 97–112.

[2] D. Boley, R. Maier and J. Kim, A parallel QR algorithm for the nonsymmetric eigenvalue problem, *Computer Physics Communications*, 53 (1989) 61–70.

[3] G. A. Geist and G. J. Davis, Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed-memory multiprocessor, *Parallel Computing*, 13 (1990) 199–209.

[4] G. Henry and R. van de Geijn, Paralllelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: myths and reality, *SIAM J. Sci. Comput.*, Vol. 17, No. 4 (1996) 870–883.

[5] G. Henry, D. Watkins, and J. Dongarra, A Parallel Implementation of the Nonsymmetric QR Algorithm for Distributed Memory Architectures, Technical Report CS-97-352, Computer Science Dept., University of Tennessee, 1997.

[6] A. Nishida, R. Suda, and Y. Oyanagi, Polynomial Acceleration for Restarted Arnoldi Iteration and its Parallelization, in J. Wang et al. ed., *Iterative Methods in Scientific Computation*, IMACS Series in Computational and Applied Mathematics, Vol. 4 (IMACS: New Brunswick, 1998) 45–52.

[7] R. Suda, A. Nishida, and Y. Oyanagi, A New Data Mapping Method for Parallel Hessenberg Double Shift QR Algorithm, *Trans. IPSJ*, Vol. 39, No. 6 (1998) 1587–1594 (in Japanese).

[8] R. A. van de Geijn, Storage schemes for Parallel Eigenvalue Algorithms, in G. H. Golub and P. van Dooren eds., *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, NATO ASI Series Vol. F70 (Springer-Verlag: Berlin 1991) 639–647.

[9] L. Wu and E. Chu, New distributed-memory parallel algorithms for solving nonsymmetric eigenvalue problems, *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing* (1995) 540–545.