Improving the Parallelism of Iterative Methods by Aggressive Loop Fusion

Jingling Xue[1], Minyi Guo[2] and Daming Wei[2]

[1] Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia
[2] School of Computer Science and Engineering, The University of Aizu, Fukushima 965-8580, Japan

Abstract. Traditionally, loop nests are fused only when the data dependences in the loop nests are not violated. This paper presents a new loop fusion algorithm that is capable of fusing loop nests in the presence of fusion-preventing anti-dependences. All the violated anti-dependences are removed by automatic array copying. As a case study, this aggressive loop fusion strategy is applied to a Jacobi solver. The performance of iterative methods is typically limited by the speed of the memory system. Fusing the two loop nests in the Jacobi solver into one reduces data cache misses, and consequently, improves the performance results of both sequential and parallel versions of the Jacobi program, as validated by our experimental results on an HP AlphaServer SC45 supercomputer.

1 Introduction

Due to the increasing performance mismatch between processors and main memories, modern computer systems are equipped with increasingly more levels of caches (e.g., three levels in the Intel IA-64 processors) to prevent performance degradation. However, caches help speed up only those programs that exhibit good data locality. For programs that do not reuse data, their execution times are limited by the poor latency and bandwidth values of the main memory. Therefore, cache-conscious programs are important for CPU-intensive applications, where the most computations are carried out inside loop nests.

There has been a great deal of work on the exploitation of cache locality for performance enhancement. For example, the design of LAPACK is influenced by efficiency considerations in the presence of caches. The main motivation of LAPACK was to recast the algorithms in EISPACK and LINPACK into blocked versions in terms of calls to BLAS [1]. In parallel with the development of LAPACK, compiler researchers have successfully automated many of the loop transformations used in LAPACK, such as loop fusion [7], loop distribution [8] and loop tiling or blocking [12, 14, 15].

However, one fundamental limitation of existing loop transformations is that they are dependence-preserving and thus inapplicable when the data dependences in the program are violated. In [16], we introduced a new loop fusion compiler algorithm that allows arbitrary loop nests with affine loop bounds and affine array subscript expressions to be fused. In the fused program, all fusion-preventing flow (i.e., true) and output dependences are eliminated by loop tiling and all fusion-preventing anti-dependences by automatic array copying. Such an aggressive loop fusion strategy has two important benefits. First, by fusing the two loop nests that cannot be fused conventionally, we are able to exploit the data reuse across the two loop nests. Second, by creating perfect loop nests that cannot be obtained conventionally, we are able to exploit the data reuse within perfect loop nests by further applying loop tiling to these perfect nests. In [16], we demonstrated that our aggressive loop fusion can improve program performance significantly on uniprocessors with cache memories. In this paper, we show that our aggressive loop fusion can also improve the performance of parallel applications running on multi-processor computer systems. Our example is an MPI program that uses the Jacobi method to solve the Helmholtz equation. Iterative solvers for partial differential equations (PDEs) such as Jacobi are typically implemented using global sweeps over the whole data set. As a result, their performance is limited by the speed of the memory system. Improving the cache performance of iterative solvers is absolutely essential to achieving good performance for these solvers on modern computer systems. We report and analyse the performance results of our Jacobi application before and after loop fusion is applied. The fused program yields improved performance due to improved data locality and also slightly reduced message communication cost.

Like Gauss-Seidel and SOR (Successive Over-Relaxation) methods, Jacobi is a classic iterative solver for PDEs. These solvers ares still important today because they are useful either as models for more complex methods or as building blocks from which more advanced methods, such as multigrid, can be constructed. This paper is not concerned with designing fast iterative solvers. Instead, the thesis of this work is that an aggressive loop fusion strategy can improve the performance of parallel applications for which the existing loop fusion is inapplicable. One future work is to apply our technique to multigrid methods.

The rest of this paper is organised as follows. Section 2 introduces an algorithm that fuses loop nests in the presence of violated anti-dependences. In Section 3, we apply this algorithm to transform a Jacobi program consisting of two loop nests into one perfect loop nest. Section 4 presents and analyses our experimental results on uniprocessor and multi-processor systems. Section 5 compares with the related work. Section 6 concludes the paper.

2 An Aggressive Loop Fusion Algorithm

We consider array-dominated programs consisting of multiple loop nests whose loop bounds and array subscript expressions are affine expressions of the surrounding loop variables. The fusion of two perfect loop nests is legal iff all dependences from the first (i.e., the lexically earlier) nest to the second nest are not reversed in the fused program [13, p. 315]. The dependences that are reversed are known as the *fusion-preventing dependences.* There are three kinds of fusion-preventing dependences: flow (i.e., write before read) dependences, output (i.e., write before write dependences) and anti- (i.e., read before write) dependences.

Suppose we are given two perfect loop nests that are to be fused by embedding the iteration space of one nest inside that of the other in a certain way. The two nests may not have the same loop bounds in a common dimension or even the same number of loops. We propose to eliminate all the fusion-preventing dependences between the two nests in two steps. We eliminate all the fusion-preventing flow and output dependences by applying loop tiling or loop shifting [4] to the first loop nest. In [16], loop tiling is used. This first step is omitted but will be illustrated in Section **??** by an example. We eliminate all the fusion-preventing anti-dependences by inserting array copy operations inside the second loop nest. This second step is the topic discussed in Section 2.2.

2.1 Example

Consider the following two perfect loop nests, where `N` is assumed to be odd:

```
L1 : do i=2, N
       do j=2, N
S1 :     a(i,j)=b(i-1)+b(i+1)
L2 : do i=2, N                          (1)
       do j=2, N
S2 :     b(i)=a(i,j-1)+a(i,j+1)
```

Suppose that one wants to fuse the two nests as follows:

```
     do i=2, N
     do j=2, N                          (2)
S1 :     a(i,j)=b(i-1)+b(i+1)
S2 :     b(i)=a(i,j-1)+a(i,j+1)
```

where the identical iteration points from both loop nests are fused together.

Dependence	From S_1	To S_2
Flow	`a(i,j)`	`a(i,j-1)`
Flow	`a(i,j)`	`a(i,j+1)`
Anti-	`b(i-1)`	`b(i)`
Anti-	`b(i+1)`	`b(i)`

Table 1. The dependences from S_1 to S_2 in (1).

To check whether the fusion is legal or not, let us examine the four dependence relations from S_1 to S_2 as summarised in Table 1. In the fused program (2), the first and last are preserved but the other two are violated.

We can eliminate the fusion-preventing flow dependence, i.e., the second flow dependence in Table 1, by applying loop tiling to $\mathcal{L}_1$. In the fused program (2), this dependence can be described by dependence vector $(0, 1)$. To avoid reversing this dependence, we tile the inner loop `j` only for S_1 by a tile size of 2 to get:

```
         do i=2, N
         do j=2, N
            if (j<=(N-1)/2+1)
              do jj=2j-2, 2j-1                          (3)
$S_1$            a(i,jj)=b(i-1)+b(i+1)
$S_2$ :     b(i)=a(i,j-1)+a(i,j+1)
```

Recall the oddness of `N` is assumed. To enhance instruction-level parallelism, the compiler typically unrolls the `jj` loop:

```
do i=2, N
do j=2, N
   if (j <= (N-1)/2+1)
     a(i,2j-2)=b(i-1)+b(i+1)                            (4)
     a(i,2j-1)=b(i-1)+b(i+1)
   b(i)=a(i,j-1)+a(i,j+1)
```

The effect of loop tiling can also be understood as follows. First, the loop `j` in $\mathcal{L}_1$ in (1) is tiled by a tile size of 2:

```
$\mathcal{L}_1'$ : do i=2, N
        do j=2, (N-1)/2+1
$S_1'$ :       do jj=2j-2, 2j-1
                 a(i,jj)=b(i-1)+b(i+1)                  (5)
$\mathcal{L}_2$ : do i=2, N
        do j=2, N
$S_2$ :        b(i)=a(i,j-1)+a(i,j+1)
```

where $\mathcal{L}_1'$ is the tiled version of $\mathcal{L}_1$ and S_1' (including the `jj` loop) is the body of $\mathcal{L}_1'$. Fusing $\mathcal{L}_1'$ and $\mathcal{L}_2$ in the same way as $\mathcal{L}_1$ and $\mathcal{L}_2$ before (i.e., the identical iterations of $\mathcal{L}_1'$ and $\mathcal{L}_2$ are fused together) yields the program given in (3).

Consider the fused program (4) again. The first anti-dependence, i.e., the third dependence listed in Table 1 is still violated. We eliminate this dependence via automatic array copying. In the following program, a 1-D array, `H`, is introduced. The copy statements are inserted in lines 1 and 8 so that the copied (i.e., correct) rather than overwritten (i.e., incorrect) values are read in lines 5 – 6:

```
1 H(1)=b(1) // array copying
2 do i=2, N
3   do j=2, N
4      if (j<=(N-1)/2+1)
5         a(i,2(j-1))=H(i-1)+b(i+1)                     (6)
6         a(i,2j-1)=H(i-1)+b(i+1)
7      if (i<=N-1 && j==2)
8         H(i)=b(i) // array copying
9      b(i)=a(i,j-1)+a(i,j+1)
```

In this final program, all the four dependence relations given in Table 1 are satisfied. In addition, the final program has the same input/output behaviour as the original program. As one single perfect loop nest, the final program can be tiled in the normal manner [13, 14].

2.2 Algorithm

In the case of multiple loop nests, our fusion strategy is applied iteratively bottom-up, starting from the last two nests. Let there be K perfect loop nests, identified by $\mathcal{L}_1, \dots, \mathcal{L}_K$, from the beginning to the end of the program:

```
$\mathcal{L}_1$:  do $I_1 = L_{1,1}, U_{1,1}$
                ⋮
                  do $I_{n_1} = L_{1,n_1}, U_{1,n_1}$
                     $BODY_1(I_1, \dots, I_{n_1})$
        ⋮                                                      (7)
$\mathcal{L}_K$: do $I_K = L_{K,1}, U_{K,1}$
                ⋮
                  do $I_{n_K} = L_{K,n_K}, U_{K,n_K}$
                     $BODY_K(I_1, \dots, I_{n_K})$
```

where the loop bounds of each loop nest are affine. Two different loop nests may not have the same loop bounds in a common dimension or even the same number of loops. Let IS_k be the n_k-dimensional iteration space of the k-th loop nest $\mathcal{L}_k$. Let $n = \max\{n_k \mid 1 \leqslant k \leqslant K\}$. If the dependences in the program (7) are ignored for the moment, it is always possible to fuse the K nests into one perfect loop nest whose n-dimensional iteration space is given by:

$$IS = \{(I_1, \dots, I_n) \mid \forall\ 1 \leqslant i \leqslant n : L_i \leqslant I_i \leqslant U_i\} \tag{8}$$

This consists of finding an injective mapping from IS_k to IS for every nest $\mathcal{L}_k$:

$$F_k : IS_k \mapsto IS \tag{9}$$

The fused program becomes one single perfect loop nest as follows:

```
do $I_1 = L_1, U_1$
    ⋮
  do $I_n = L_n, U_n$
     if $(I_1, \dots, I_n) \in F_1(IS_k)$
        $BODY_1(F_1^{-1}(I_1, \dots, I_n))$                    (10)
            ⋮
     if $(I_1, \dots, I_n) \in F_K(IS_K)$
        $BODY_K(F_K^{-1}(I_1, \dots, I_n))$
```

where all original K loop nests “share” the same iteration vector $I = (I_1, \dots, I_n)$.

Figure 1 illustrates that there are many different ways of fusing a sequence of loop nests into a single loop nest. This paper is not concerned with finding the best among all solutions for a given program. However, our aggressive loop fusion algorithm works for any fused program thus obtained.

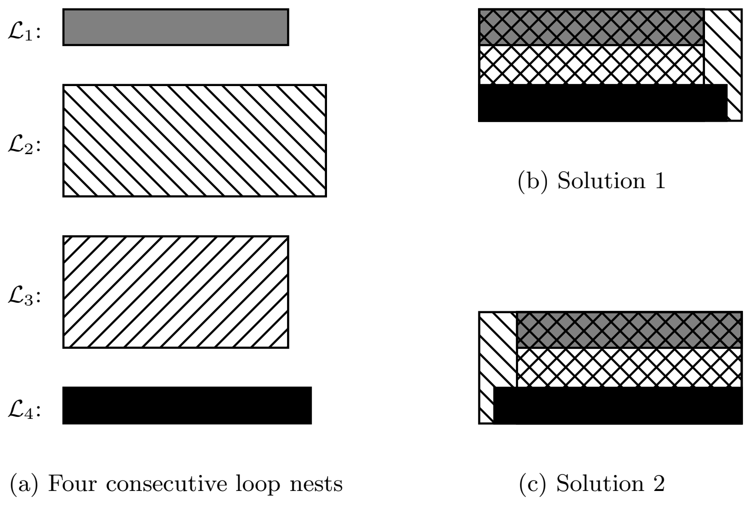


Fig. 1. A geometrical illustration for two of many different ways of fusing four loop nests. The iteration spaces of the four loop nests are depicted as rectangles.

The loop fusion used for transforming the original program (7) to the fused program (10) are illegal when some dependences in the original program (7) are violated. Figure 2 gives an algorithm for eliminating all the fusion-preventing anti-dependences so that both programs have the same input/output behaviour. As we mentioned earlier, we assume that the violated flow and output dependences have already been eliminated by some other means such as loop tiling [16] and/or loop shifting [4].

Our algorithm makes use of the following notations. A denotes an arbitrary but fixed array in the original program (7), which may be accessed in all its K loop nests, $\mathcal{L}_1, \dots, \mathcal{L}_K$. All p_k read references of A in $\mathcal{L}_k$ are identified by integers consecutively, starting from 1. Thus, a read reference identified by s signifies that it is the s-th read reference accessed among all p_k read references. Let $\mathit{Reads}_A(k) = \{1, \dots, p_k\}$. Similarly, $\mathit{Writes}_A(k) = \{1, \dots, q_k\}$ denotes the set of all q_k write references in $\mathcal{L}_k$. $S_A^{k,s}$ denotes the set of iterations at which the s-th read or write reference is accessed and $f_A^{k,s}(I)$ its array subscript expression, where $I = (I_1, \dots, I_n)$ is the iteration vector of the fused program (10).

```
1   **ALGORITHM:** $\mathit{ElimRW}(P)$
2   **INPUT:** A program $P$ of the form given in (7)
3   **OUTPUT:** A fused program $P'$ with same input/output behavior as $P$
4   Let $P'$ be the fused program (10) obtained from $P$
5   for every array $A$ in the program $P$
6       for $k = K-1, 1$
7           for $k' = k+1, K$
8               Compute $RW_A(k,k')$
9           $\overline{RW}_A(k) := \bigcup_{k'=k+1}^{K} \{(I', k', s') \mid (I, I', s') \in RW_A(k,k')\}$
10          Compute $\min_{\prec} \overline{RW}_A(k)$
11          Introduce a new array for $A$, $H_{A,k}$, of size $|\min_{\prec} \overline{RW}_A(k)|$ in $P'$
12          Insert the copy operations at the beginning of $\mathcal{L}_{k+1}$'s loop body in $P'$

                if $(I, k', s') \in \min_{\prec} \overline{RW}_A(k)$
                    $H_{A,k}(f_{k',s'}(I)) = A(f_{k',s'}(I))$

13          for read reference $s \in \mathit{Reads}_A(k)$, i.e., $A(f_A^{k,s}(I))$ in $\mathcal{L}_k$
14              $C_A^{k,s} := \{I \mid I \in S_A^{k,s} \wedge\ k' > k \wedge s' \in \mathit{Writes}_A(k') \wedge I' \in S_A^{k',s'}$
                    $\wedge\ I' \prec I \wedge f_A^{k,s}(I) = f_A^{k',s'}(I')\}$
15              Replace $A(f_A^{k,s}(I))$ by:

                if $I \in C_A^{k,s}$
                    $H_{A,k}(f_A^{k,s}(I))$
                else
                    $A(f_A^{k,s}(I))$
```

Fig. 2. An algorithm for fixing all the fusion-preventing data dependences.

Consider two loop nests $\mathcal{L}_k$ and $\mathcal{L}_{k'}$, where $k < k'$. $RW_A(k,k')$ is the set of anti-dependences of A that prevent $\mathcal{L}_k$ and $\mathcal{L}_{k'}$ from being fused:

$$\begin{aligned} RW_A(k,k') = \{(I, I', s') \mid\ & s \in \mathit{Reads}_A(k) \wedge I \in S_A^{k,s} \\ & \wedge\ s' \in \mathit{Writes}_A(k') \wedge I' \in S_A^{k',s'} \\ & \wedge\ I' \prec I \wedge f_A^{k,s}(I) = f_A^{k',s'}(I')\} \end{aligned} \tag{11}$$

where $\prec$ denotes the lexicographhc "less than" order between iteration vectors.

To eliminate the violated anti-dependences from $\mathcal{L}_k$ to $\mathcal{L}_{k'}$, where $k < k'$, we insert array copy operations to copy the values of A just before they are incorrectly overwritten by a write reference in Writes_A so that all read references in Reads_A can be modified to access the original values of A correctly.

Let us explain the basic idea behind our algorithm *ElimRW* given in Figure 2. Here RW stands for Read before Write dependences. Given the fused program (10), we eliminate all the violated anti-dependences iteratively bottom-up across the K loop nests starting from the last two loop nests $\mathcal{L}_{K-1}$ and $\mathcal{L}_K$. First, we eliminate all the violated anti-dependences from $\mathcal{L}_{K-1}$ to $\mathcal{L}_K$. Next, we eliminate all the violated anti-dependences from $\mathcal{L}_{K-2}$ to $\mathcal{L}_{K-1}$ and $\mathcal{L}_K$. This process is repeated until $\mathcal{L}_1$ is processed, in which case, we eliminate all the violated anti-dependences from $\mathcal{L}_1$ to the last $n-1$ nests from $\mathcal{L}_2$ through $\mathcal{L}_K$.

ElimRW takes as input a program P of the form (7) and produces as output a fused program P' that has the same input/output behavior as P. In line 4, we obtain the fused program P' of the form (10) from P as discussed earlier. In line 5, we process all arrays in the program, one by one, in any order. In the **for** loop starting at line 6, we eliminate iteratively all violated anti-dependences bottom-up across all K loop nests. During the k-th iteration of this **for** loop, we aim at eliminating all the fusion-preventing anti-dependences from $\mathcal{L}_k$ to $\mathcal{L}_{k+1}, \ldots, \mathcal{L}_K$. In lines 7 – 9, $\overline{RW}_A(k)$ is calculated to be the set of all such violated anti-dependences. To insert the required copy operations correctly, we must know the earliest iteration at which a particular anti-dependence is violated. The set of all these earliest points is given by $\min_{\prec} \overline{RW}_A(k)$ in line 10, where the iteration vector I is treated as a parameter and the iteration vector I' as a variable. If all constraints involved in defining $\overline{RW}_A(k)$ are affine expressions of I' and I, $\min_{\prec} \overline{RW}_A(k)$ can be computed parametrically (in terms of I) using the PIP [5] or Omega Calculator [10] (both tools) are based on integer programming).

By definition, $\min_{\prec} \overline{RW}_A(k)$ contains the earliest writes at which some anti-dependences are violated in the program P. In lines 11 – 12, we insert the copy statements to copy the old values of A at these iterations just before they are overwritten. In lines 13 – 15, we make sure that the copied values are used correctly only at the iterations defined by the predicate $C_A^{k,s}$ in line 14.

Note that the correctness of *ElimRW* relies on the fact that all the fusion-preventing flow and output dependences have been eliminated first.

Theorem 1. *The input program P to and the output program P' from ElimRW have the same input/output behaviour.*

Proof. As a loop invariant at the beginning of the k-th iteration of the **for** loop in line 6, all the violated anti-dependences in $\overline{RW}_A(k+1), \ldots, \overline{RW}_A(K)$ have been eliminated. During the k-th iteration, the violated anti-dependences in $\overline{RW}_A(k)$ are all eliminated by array copying. In addition, the copy array, $H_{A,k}$, introduced in line 10 will not affect the values in the copy arrays, $H_{A,k+1}, \ldots, H_{A,K}$, that may have been introduced in the earlier iterations of the **for** loop in line 6. $\square$

The number of copying arrays introduced for an existing array depends only on the number of fused loop nests. If array expansion [6] is used to eliminate output and anti-dependences, the amount of extra space introduced often depends on the problem size. For example, a 2-D array of size $N \times N$ is often expanded into a 3-D array of size $N \times N \times N$. In our case, the worst-case scenario is $N \times N \times L$, where L is the number of loop nests in the program.

3 A Jacobi Program

Figure 3 gives a Fortran90 program for solving the Helmholtz equation on a regular mesh, using an iterative Jacobi method with over-relaxation. The program is taken from [2] except that the roles of `u` and `unew` are swapped. There are two loop nests in the `while`, i.e., the time loop. The two-dimensional array `u` is used to store the results of the previous iteration and the two-dimensional array `unew` is used to store the results of the current iteration. In the first loop nest, the sweep operation is executed, including the sum of the squared residuals used for the error estimation and the termination condition of the surrounding `while` loop. In the second loop nest, `unew` is copied to `u`.

The two loop nests in the `while` loop cannot be fused by the conventional loop fusion transformation because the cross-nest anti-dependences from the two read references `u(i-1,j)` and `u(i,j-i)` in the first loop nest to the write reference `u(i,j)` will be violated. Therefore, the inter-nest data reuse for the two arrays cannot be exploited for a reasonably large mesh.

We can apply *ElimRW* to fuse the two loop nests legally as follows. The input program P consists of the two loop nests in the Jacobi program. In line 4, we obtain the fused loop nest, P', as depicted in Figure 4. There is only one variable, `u`, whose anti-dependences may be violated. So the **for** loop in line 5 has only one iteration. There are only two nests. So $K = 2$. The **for** loop in line 6 also executes for only one iteration. Let the four read references of `u` in the first nest be numbered as $\texttt{u(i-1,j)}^1$, $\texttt{u(i+1,j)}^2$, $\texttt{u(i,j-1)}^3$ and $\texttt{u(i,j+1)}^4$. So $\mathit{Reads}_{\texttt{u}} = \{1, 2, 3, 4\}$. There is only one write reference, `u(i,j)`, in the second loop nest. So $\mathit{Writes}_{\texttt{u}} = \{1\}$. We note that all anti-dependences from $\texttt{u(i+1,j)}^2$ and $\texttt{u(i,j+1)}^4$ to `u(i,j)` are respected. But all the anti-dependences from $\texttt{u(i-1,j)}^1$ and $\texttt{u(i,j-1)}^3$ to `u(i,j)` are violated. In line 8, we obtain:

$$\begin{aligned}
\mathit{RW}_{\texttt{u}}(1,2) = \{((j',i'),(j,i),1) \mid{} & 2 \leqslant j,j' \leqslant m-1 \wedge 2 \leqslant i,i' \leqslant n-1 \\
& \wedge (j',i') \prec (j,i) \wedge ((j',i') = (j-1,i) \vee (j',i') = (j,i-1))\} \\
= \{((j',i'),(j,i),1) \mid{} & 2 \leqslant j,j' \leqslant m-1 \wedge 2 \leqslant i,i' \leqslant n-1 \\
& \wedge ((j',i') = (j-1,i) \vee (j',i') = (j,i-1))\}
\end{aligned} \tag{12}$$

In line 9, we have $\overline{\mathit{RW}}_{\texttt{u}}(1) = \mathit{RW}_{\texttt{u}}(1,2)$. In the fused program given in Figure 4, all elements of `u` except those in row `n-1` and column `m-1` are written too earlier before their values have been actually consumed by $\texttt{u(i-1,j)}^1$ and $\texttt{u(i,j-1)}^3$.

To fix these violated anti-dependences, we compute $\min_{\prec} \overline{\mathit{RW}}_{\texttt{u}}(1)$ in line 10. In this case, we actually have $\min_{\prec} \overline{\mathit{RW}}_{\texttt{u}}(1) = \overline{\mathit{RW}}_{\texttt{u}}(1)$. The subscript expression for `u(i.j)` is $f^{2,1}_{\texttt{u}}(i,j) = (i,j)$. According to lines 11 – 12, we introduce a new array, `H`, and insert the following copy statement just before `u(i,j)=unew(i,j)`:

```
if (j .ne. m-1 .and. i .ne. n-1) then
   H(i,j)=u(i,j)
end if
```
(13)

where the if conditional is obtained from the specifying constraints of $\min_{\prec} \overline{\mathit{RW}}_{\texttt{u}}(1)$ simplified under the context $2 \leqslant j \leqslant m-1 \wedge 2 \leqslant i \leqslant n-1$, which defines the iteration space of the fused loop nest in Figure 4.

In lines 13 – 15, we need to examine all the four references $\texttt{u(i-1,j)}^1$, $\texttt{u(i+1,j)}^2$, $\texttt{u(i,j-1)}^3$ and $\texttt{u(i,j+1)}^4$ to see how they should be modified to

```
      subroutine jacobi (n,m,dx,dy,alpha,omega,u,f,tol,maxit)

      double precision dx,dy,f(n,m),u(n,m),alpha, tol,omega
      double precision error,resid,ax,ay,b
      double precision unew(n,m)

      ax = 1.0/(dx*dx) ! X-direction coef
      ay = 1.0/(dy*dy) ! Y-direction coef
      b = -2.0/(dx*dx)-2.0/(dy*dy) - alpha ! Central coeff

      error = 10.0 * tol
      k = 1

      do while (k.le.maxit .and. error.gt. tol)
         error = 0.0
         do j = 2,m-1
           do i = 2,n-1
             resid = (ax*(u(i-1,j) + u(i+1,j)) &
&              + ay*(u(i,j-1) + u(i,j+1)) &
&              + b * u(i,j) - f(i,j))/b
             unew(i,j) = u(i,j) - omega * resid
             error = error + resid*resid
           end do
         enddo

         do j=2,m-1
           do i=2,n-1
             u(i,j) = unew(i,j)
           enddo
         enddo

         k = k + 1
         error = sqrt(error)/dble(n*m)

      enddo ! End time loop

      print *, 'Total Number of Iterations ', k
      print *, 'Residual ', error

      maxit = k - 1

      return
      end
```

Fig. 3. A Jacobi program for solving the Helmholtz equation.

```
      subroutine jacobi (n,m,dx,dy,alpha,omega,u,f,tol,maxit)

      double precision dx,dy,f(n,m),u(n,m),alpha, tol,omega
      double precision error,resid,ax,ay,b
      double precision unew(n,m)

      ax = 1.0/(dx*dx) ! X-direction coef
      ay = 1.0/(dy*dy) ! Y-direction coef
      b = -2.0/(dx*dx)-2.0/(dy*dy) - alpha ! Central coeff

      error = 10.0 * tol
      k = 1

      do while (k.le.maxit .and. error.gt. tol)
         error = 0.0
         do j = 2,m-1
           do i = 2,n-1
             resid = (ax*(u(i-1,j) + u(i+1,j)) &
&              + ay*(u(i,j-1) + u(i,j+1)) &
&              + b * u(i,j) - f(i,j))/b
             unew(i,j) = u(i,j) - omega * resid
             error = error + resid*resid
             u(i,j) = unew(i,j)
           enddo
         enddo

         k = k + 1
         error = sqrt(error)/dble(n*m)

      enddo ! End time loop

      print *, 'Total Number of Iterations ', k
      print *, 'Residual ', error

      maxit = k - 1

      return
      end
```

Fig. 4. The code obtained by fusing the two loop nests given in Figure 3.

read the copied values in H. We find that $C_{\mathtt{u}}^{1,2} = C_{\mathtt{u}}^{1,4} = \emptyset$, meaning that the anti-dependences originating from the second and fouth read references are not violated. However, $C_{\mathtt{u}}^{1,1} = C_{\mathtt{u}}^{1,3} \neq \emptyset$. Under the context $2 \leqslant j \leqslant m-1 \wedge 2 \leqslant i \leqslant n-1$, the specifying constraint for $C_{\mathtt{u}}^{1,1}$ is simplified to $i \geqslant 3$ and that for $C_{\mathtt{u}}^{1,3}$ to $j \geqslant 3$.

Therefore, in line 15, the read reference `u(i-1,j)`^{1} should be replaced by:

```
if (i .ge. 3) then
   H(i-1,j)
else
   u(i-1,j)
end if
```
(14)

Similarly, the read reference `u(i,j-1)`^{1} should be replaced by:

```
if (j .ge. 3) then
   H(i,j-1)
else
   u(i,j-1)
end if
```
(15)

In practice, if we choose to copy redundantly some boundaries elements of an array, then the if conditionals like those in (13 – (15) can often be simplified or even completely eliminated. Under such optimisations, which can be incorporated into *ElimRW*, we obtain the final fused version of our Jacobi program shown in Figure 5. By choosing to copy row `n-1` and column `m-1` redundantly, the if conditional in (13) is removed. Similarly, by copying row 1 and column 1 redundantly just before the `while` loop, the if conditionals in (14) and (15) have been removed. Note that the array `unew` is no longer needed. So the access `unew(i,j)` has been replaced by a scalar, `tmp`. The copy array `H` has the same size as `unew`. In this example, loop fusion has not caused any extra memory space increase.

In the final program, the two arrays `u` and `H` are accessed within a single loop nest. Therefore, their data elements exhibit better data reuse in cache memories.

4 Experiments

We evaluate this work using the Jacobi example on a 126-node HP AlphaServer SC45 supercomputer. Each node has four 1GHz ev68 (Alpha 21264C) CPUs running OSF1 sc0 V5.1. Each CPU has a 64KB (on-chip) write back and write allocate data cache with FIFO replacement policy. The L1 data cache is 2-way set-associative with a cache line size of 64B. Each CPU also has an (off-chip) L2 unified cache, which is direct-mapped and has a capacity of 8MB. Each node has between 4GB and 16GB of RAM and between 2 and 6 36GB SCSI disks. Due to the use of the fat-tree interconnect of the Quadrics "Elan3" network, the SC45 computer system achieves an MPI latency of less than 5 usecs and an MPI bandwidth of 250 Mbytes/sec (bi-directional).

In all our experiments, `maxit=1000` is fixed and the `while` loop has always completed in exactly 1000 iterations. The regular mesh on which the Jacobi method operates is defined by two problem size parameters, `m` and `n`. In all our

```
      subroutine jacobi (n,m,dx,dy,alpha,omega,u,f,tol,maxit)

      double precision dx,dy,f(n,m),u(n,m),alpha, tol,omega
      double precision error,resid,ax,ay,b
      double precision unew(n,m)
      double precision H(n,m)

      ax = 1.0/(dx*dx) ! X-direction coef
      ay = 1.0/(dy*dy) ! Y-direction coef
      b = -2.0/(dx*dx)-2.0/(dy*dy) - alpha ! Central coeff

      error = 10.0 * tol
      k = 1

      do j=2,m-1
        H(1,j) = u(1,j)
      enddo

      do i=2,n-1
        H(i,1) = u(i,1)
      enddo

      do while (k.le.maxit .and. error.gt. tol)
         error = 0.0
         do j = 2,m-1
           do i = 2,n-1
             resid = (ax*(H(i-1,j) + u(i+1,j)) &
   &           + ay*(H(i,j-1) + u(i,j+1)) &
   &           + b * u(i,j) - f(i,j))/b
             H(i,j) = u(i,j)
             tmp = u(i,j) - omega * resid
             error = error + resid*resid
             u(i,j) = tmp
           enddo
         enddo

         k = k + 1
         error = sqrt(error)/dble(n*m)

      enddo ! End time loop

      print *, 'Total Number of Iterations ', k
      print *, 'Residual ', error

      maxit = k - 1

      return
      end
```

Fig. 5. Final code from *ElimRW* with all violated anti-dependences of u fixed.

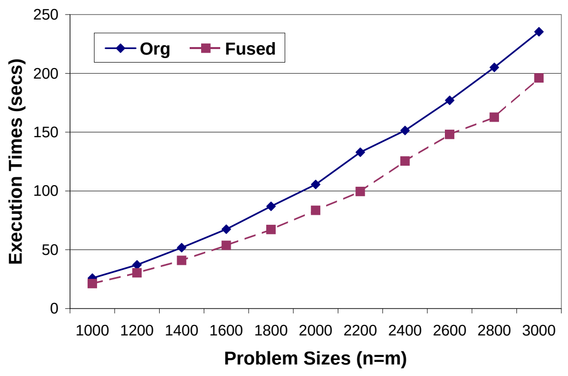


Fig. 6. The execution times of Org and Fused.

experiments, a square mesh is used: `n=m`. All arrays are of double precision. So an array of size 90×90 fills up roughly the 64KB L1 data cache and an array of size 1024×1024 fills up exactly the 8MB L2 cache for the Alpha 21264 CPU.

In Section 4.1, we discuss our experimental results on a single CPU. In Section 4.2, we discuss our experimental results on multi-processor platforms.

4.1 Uniprocessors

There are two sequential programs, Org and Fused, where Org is the original program given in Figure 3 and Fused denotes the fused program shown in Figure 5. We demonstrate the performance benefits of our aggressive loop fusion algorithm using the Jacobi example on a single 21264 CPU. Both programs are compiled by the HP Fortran90 compiler (V5.5A) at the optimisation level "-fast".

Figure 6 compares the execution times of Org and Fused. The speedups of fused program Fused over Org range from 19.62% to 29.27% with an average of 24.38%. Figure 7 compares the L1 data cache misses of both programs. The cache misses are estimated using the DineroIV cache simulator for the array accesses only. In Org, the inter-nest data reuse for the two arrays `u` and `unew` cannot be exploited. By fusing the two loop nests, the single loop nest in Fused also contains two arrays of the same size. But better data reuse for the two arrays can now be exploited. As a result, we observe some significant reductions in the L1 cache misses across all the problem sizes used. In comparison with the original program Org, Fused enjoys an average of 40% L1 cache miss reduction for the problem sizes simulated. The decreases in cache misses have translated into the performance improvements as shown in Figure 6.

4.2 Multiple Processors

The MPI versions of sequential programs Org and Fused are obtained using a 1D domain decomposition. This choice is made primarily to facilitate a simple

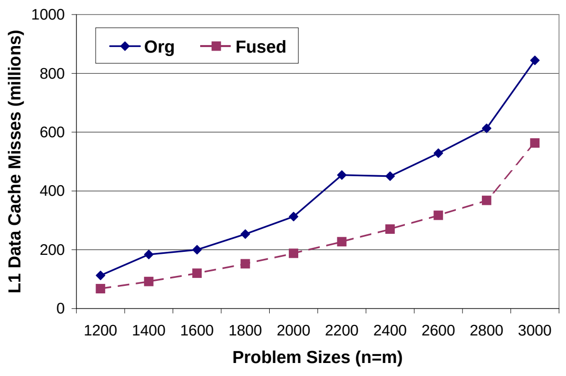


Fig. 7. The simulated L1 data cache misses of Org and Fused.

boundary condition implementation. Suppose that $\mathcal{P}$ processors are available. The regular mesh `n x m` is divided into $\mathcal{P}$ vertical strips, with one being allocated to one processor. In other words, the columns of each array are blocked distributed among the $\mathcal{P}$ processors. As a result, the part of the global array `u(n,m)` allocated to the p-th processor, where $0 \leqslant p < \mathcal{P}$, is `u(n,mlo:mhi)`, where $\texttt{mlo} = p \times (\texttt{m} - 2)/\mathcal{P} + 1$ and $\texttt{mhi} = \min(p + 1) \times (\texttt{m} - 2)/\mathcal{P} + 2, \texttt{m})$. The array `unew(n,m)` in the program Org and the array `H(n,m)` in the program Fused are both distributed in the same manner.

The processor p is responsible for computing the values for the sub-mesh `n x (mlo+1:mhi-1)`. During each iteration of the `while`, i.e., the time loop, the processor p first sends asynchronously column `mlo+1` to its left neighbouring processor $p - 1$ and column `mhi-1` to its right neighbouring processor $p + 1$. In addition, the processor receives synchronously column `mlo` from its left neighbouring processor $p - 1$ and column `mhi` from its right neighbouring processor $p+1$. Only after having received both columns, can the processor p start working on its allocated columns. At the end of each `while` loop, `MPI_ALLREDUCE` is called to calculate the error for the current iteration.

The MPI versions of Org and Fused are referred to as Org-MPI and Fused-MPI, respectively. Both programs are compiled by the HP Fortran90 compiler (V5.5A) on the SC45 supercomputer at the optimisation level "-fast". The SC45 uses a version of MPI that is based on MPICH 1.2.4. In this particular supercomputer, we are allowed to use a maximum of 60 CPUs. In all our experiments on MPI applications, a regular mesh of 5000×5000 is used. As before, we set `maxit=1000` so that the `while loop` runs for exactly 1000 iterations in our experiments.

Figure 8 compares the execution times of Org-MPI and Fused-MPI. Figure 9 shows the performance improvements of Fused-MPI over Org-MPI. The performance improvements range from 12.85% to 27.74% with an average of 19.35%. Figure 10 illustrates quantitatively how the improvements in cache locality have contributed to the overall speedups of our example application. For each pro-

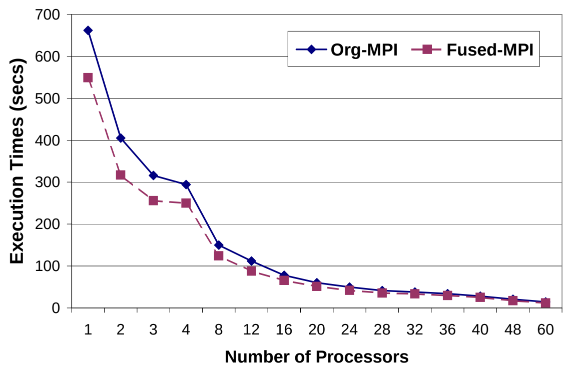


Fig. 8. The execution times of Org-MPI and Fused-MPI.

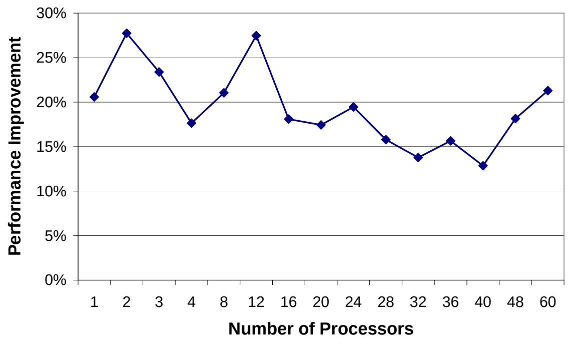


Fig. 9. The performance improvements of Fused-MPI over Org-MPI.

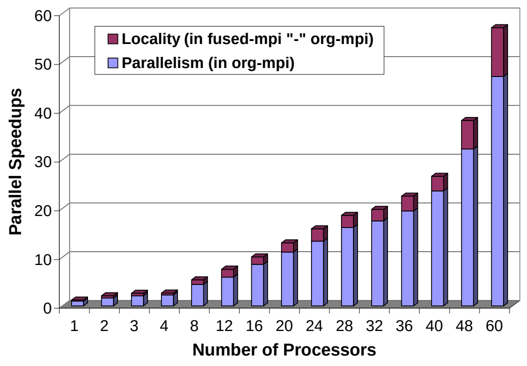
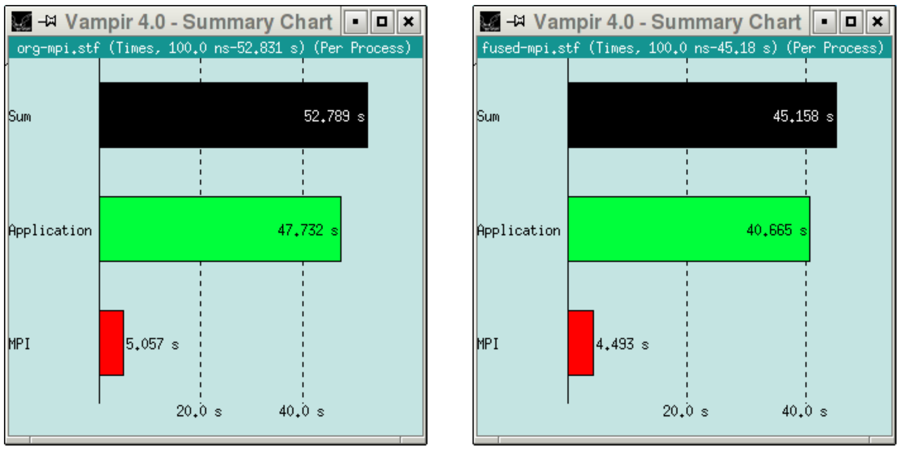


Fig. 10. Effects of improving cache locality in Fused-MPI on parallel speedups

cessor configuration, the bottom bar represents the parallel speedup of Org-MPI over Org and the entire bar the parallel speedup of Fused-MPI over Org. Therefore, the top bar represents the increase in the parallel speedup (in absolute terms) due to the improved cache locality. These increases range from 0.21 to 7.66 with an average of 2.38 for the processor configurations used.



(a) Org-MPI

(b) Fused-MPI

Fig. 11. Performance analysis of Org-MPI and Fused-MPI when $\mathcal{P} = 24$.

We have also compiled and linked Org-MPI and Fused-MPI with Vampirtrace 4.0 and analysed the performance results of both programs using Vampir. Figure 11 shows the summary charts for both programs in the 24-processor configuration. By performing loop fusion aggressively, we have reduced not only the computation time but also slightly the communication time for the Jacobi program. Since Fused-MPI exhibits better data reuse than Org-MPI, each processor completes its allocated computations earlier. This may reduce the idle time that the processors spend on waiting for messages. Therefore, the overall communication time in Fused-MPI is slightly reduced compared to Org-MPI. Note that Vampirtrace does incur some instrumentation overhead. So the execution times shown in Figure 11 are not exactly the same as those shown in Figure 8.

5 Related Work

Loop fusion is a standard compiler optimisation employed in a number of research and commercial compilers. Some earlier work on the topic can be found in [3, 9, 13] and the references therein. However, loop fusion is applicable only when the dependences in the program are not violated. In [16], we presented the first algorithm that allows arbitrary affine loop nests to be fused in the presence of the fusion-preventing flow, output and anti-dependences. The motivation of our earlier work was to improve the cache performance of sequential programs on uniprocessors. In this paper, we investigate the performance benefits of this aggressive loop fusion algorithm for parallel applications.

Many scientific and engineering applications require the solution of partial differential equations (PDEs). A common approach discretises the input domain, thereby transforming a PDE problem into one of solving a linear system. For large systems with several millions of unknowns, the methods of choice are all iterative. Classic iterative solvers are Jacobi, Gauss-Seidel and SOR (Successive Over-Relaxation) methods. These solvers remain important because they are useful either as models for more complex methods or as building blocks from which more advanced methods, such as multigrid, can be constructed.

However, iterative methods do not exhibit good data reuse since they are typically implemented using global sweeps over the whole data set. Song and Li [11] describe special-purpose techniques for tiling Jacobi-like codes to achieve good performance improvements on uniprocessors. In this paper, we show that fusing the loop nests in Jacobi-like codes can achieve good performance results on both uniprocessor and multi-processor systems.

6 Conclusion

This paper presents a loop fusion algorithm that is capable of fusing loop nests even when the conventional loop fusion optimisation fails. In the presence of fusion-preventing anti-dependences, we eliminate all these violated dependences by means of automatic array copying. We assume that all violated flow and output dependences have been eliminated before our algorithm is applied. In [16], we demonstrated that such an aggressive loop fusion strategy achieves good performance improvements on uniprocessors with cache memories. Taking a Jacobi program as an example, we show in this paper that such a strategy is also effective for improving the performance of MPI applications on multi-processor systems. In general, the performance of stencil codes is limited by the speed of the memory system. Our experimental results indicate that better performance results for stencil codes can be obtained if the data reuse in these codes is improved. One future work is to investigate the performance benefits of our technique for more advanced methods such as multigrid. How to effectively combine loop fusion and loop tiling for multigrid methods is another interesting topic.

6.1 Acknowledgments

This work is supported by an ARC Grant DP0452623.

References

1. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 3rd edition, 1999.
2. The OpenMP Architecture Review Boards (ARB). http://www.openmp.org.
3. A. Darte. On the complexity of loop fusion. *Parallel Computing*, 29(6):1175 – 1193, 2000.
4. Alain Darte and Guillaume Huard. Loop shifting for loop compaction. *International Journal of Parallel Programming*, 28(5):499–534, 2000.
5. P. Feautrier. Parametric integer programming. *Operations Research*, 22:243–268, 1988.
6. P. Feautrier. Dataflow analysis for array and scalar references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.
7. Guang R. Gao, R. Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, London, UK, 1993. Springer-Verlag.
8. Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 407–416, Washington, DC, USA, 1990. IEEE Computer Society.
9. N. Manjikian and T. S. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Trans. on Parallel and Distributed Systems*, 8(2):193–209, Feb. 1997.
10. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 35(8):102–114, Aug. 1992.
11. Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, May 1999.
12. M. J. Wolfe. More iteration space tiling. In *Supercomputing '88*, pages 655–664, Nov. 1989.
13. M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.
14. J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.
15. J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Boston, 2000.
16. J. Xue, Q. Huang, and M. Guo. Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences. In *International Conference on Parallel Processing*, 2005.