Towards a Fast Parallel Sparse Matrix-Vector Multiplication

Roman Geus [a,1] Stefan Röllin [b,2]

[a] *Institute of Scientific Computing, ETH Zürich*

[b] *Integrated Systems Laboratory, ETH Zürich*

Abstract

The sparse matrix-vector product is an important computational kernel that runs ineffectively on many computers with super-scalar RISC processors. In this paper we analyse the performance of the sparse matrix-vector product with symmetric matrices originating from the FEM and describe techniques that lead to a fast implementation. It is shown how these optimisations can be incorporated into an efficient parallel implementation using message-passing.

We conduct numerical experiments on many different machines and show that our optimisations speed up the sparse matrix-vector multiplication substantially.

Key words: sparse matrices, matrix-vector multiplication, source code optimisation, parallel linear algebra

1 Performance analysis of the sparse matrix-vector product

In this paper we focus on large symmetric sparse matrices, that do not fit into the memory cache. While our matrices are stored in *symmetric sparse skyline format* (SSS), our ideas can be applied to general sparse matrices stored in other formats as well.

The SSS format [1] is an extension to the commonly used *compressed sparse row format* (CSR). For matrices in SSS format the strictly lower triangle is stored in CSR format (arrays `ia, ja, va`). The double precision array `va` of length n_{nz} contains the non-zero values of the strictly lower triangle, stored row by row. The

[1] Email: geus@inf.ethz.ch
[2] Email: roellin@iis.ee.ethz.ch

integer array ja of length n_{nz} contains the column indices of the nonzero elements as stored in the array va. The integer array ia of length $n + 1$ contains pointers to the beginning of each row in the arrays va and ja. The diagonal is stored separately in a double precision array da of length n, where n is the order of the matrix and n_{nz} is the number of non-zero values in the strictly lower triangle.

Algorithm 1 shows a matrix-vector multiplication code $y := Ax$ for a matrix stored in SSS format. The accesses to the matrix data structure are in a stride-1 loop, the access pattern on x and y is irregular and depends on the sparsity structure of A.

Algorithm 1

```
for (i = 0; i < n; i ++) {    /* loop over rows of lower triangle */
  xi = x[i];                     /* load x[i] */
  s = 0.0;
  k2 = ia[i+1];
  for (k = ia[i]; k < k2; k ++){ /* loop over nonzero elems of row*/
    j = ja[k];                   /* load column index j */
    v = va[k];                   /* load matrix element A[i,j] */
    s = s + v*x[j];              /* s = s + A[i,j] * x[j] */
    y[j] = y[j] + v*xi;          /* y[j] = y[j] + A[j,i] * x[i] */
  }
  y[i] = da[i]*xi + s;           /* y[i] = A[i,i] * x[i] + s */
}
```

To compute an upper bound for the performance of the sparse matrix-vector product for a given architecture, profound knowledge of the design of the processor and the memory subsystem is required.

In [2] it is shown how the optimal out-of-cache performance of the daxpy() Routine can be computed for the HP PA-8500 architecture. For this computation one needs to know the cache line size, the cache miss penalty and how many outstanding memory requests the processor supports. To compute the optimal out-of-cache performance of the sparse matrix-vector product in this manner is much more difficult as its performance depends on the sparsity pattern of the matrix.

Our straight-forward approach is more portable and gives comparable results. We compute the ratio η of the number of floating point operations to the number of bytes of memory traffic. For the best case scenario where x and y are read only once from memory and then kept in-cache we get $\eta \approx 0.31$ flops/byte for our test matrices described in Tab. 2. For this approximation we assume a very large write-back cache and double precision arithmetic. If we multiply η by the memory bandwidth of the system we get an upper bound of the performance of the sparse matrix-vector product.

We determine the memory bandwidth by benchmarking highly optimised computational kernels tuned for the given hardware. For this benchmark we use the vendor

System	**Processor**	**Measured Bandwidth** Mbytes/s	**Max. Perf.** Mflops/s	**Measured Perf.** Mflops/s
Intel Linux PC	500MHz Pentium III	261.44	81.10	32.50
Sun Enterprise 3500	336MHz Ultra SPARC	248.71	76.52	28.59
DEC Workstation	500MHz Alpha 21164	245.74	75.61	42.66
HP X-Class	180MHz PA-8000	558.14	171.74	48.22
HP V-Class	440MHz PA-8500	552.14	169.89	75.02
IBM SP2	160MHz POWER2 SC	1165.05	358.49	56.33
Intel Paragon	50MHz i860 XP	N/A	N/A	N/A

Table 1
Machines used for the numerical experiments, their measured memory bandwidth β, the predicted maximal performance r_{opt} and the measured performance r_{act} of Alg. 1. $\beta\eta =: r_{\text{opt}} \geq r_{\text{act}}$. *The numbers in the last column are measured using the matrix* cav2 *(see Tab. 2). Because the Intel Paragon at ETH Zürich was discontinued during the time we conducted our experiments, we don't have all results available for that machine.*

supplied BLAS `daxpy` and other routines that have a similar ratio of read to write operations as the sparse matrix-vector product. This gives more realistic results than using the peak memory bandwidth reported by the vendor.

From Tab. 1 one can observe that the measured performance of the sparse matrix-vector multiplication code is far below the optimal performance, which is limited only by the memory bandwidth. The compiler is unable to generate efficient code, mainly because of data dependencies and irregular loops. Additional cache misses generated by accesses on x and y further degrade performance. On the other hand the upper bound computed for the IBM SP2 is unrealistic because its memory cache is too small (128 KBytes) to keep the vectors x and y in-cache.

For all numerical experiments published in this paper we used the standard vendor supplied C compilers with all (safe) optimisations turned on. On the Intel Linux PC we used the GNU C compiler for our benchmarks.

2 Design of a fast sparse matrix vector product for one processor

We applied three techniques to improve the implementation of Alg. 1.

2.1 Software Pipelining

By use of *software pipelining* (reorganising the source code in such a way that the processor pipelines are better filled) we are able to load data into registers earlier (data prefetching) and reduce data-dependencies in the innermost loop iteration. This increases the instruction level parallelism.

Most compilers are unable to perform these optimisations in a satisfactory way. Often compilers do not have the information necessary to move up load instructions safely. Due to the lack of type information, they have to generate code conservatively. Also most compilers have trouble unrolling more complex loops, e.g., revisiting Alg. 1, none of the vendor supplied compilers we used were able to unroll the inner loop over `k`.

Alg. 2 is an optimized version of Alg. 1. Here all data is loaded from memory one loop iteration before it is actually needed, such that the processor can better overlap computation and memory transfers.

Algorithm 2

```
for (i = 0; i < n; i ++) {
  /* initialization and mult with diagonal element */
  xi = x[i]; di = da[i];
  s = 0.0; k2 = ia[i+1]; yi = di*xi;
  if (k < k2) {
    /* first iteration: prefetch data  */
    j = ja[k]; v = va[k]; yj = y[j];
    k ++;
    while (k < k2) {
      /* prefetch j and v for next iteration */
      j_ = ja[k]; v_ = va[k];
      /* calc using prefetched data */
      s += v*x[j];
      y[j] = yj + v*xi;
      /* prefetch y for next iteration */
      yj = y[j_];
      /* "rename" prefetched data */
      j = j_; v = v_;
      k ++;
    }
    /* last iteration: no prefetch
    s += v*x[j];
    y[j] = yj + v*xi;
  }
  y[i] = yi + s;
}
```

2.2 Register Blocking

We reduce the number of memory accesses by *register blocking*, i.e. splitting the matrix A into a sum of matrices $A = A_1 + \ldots + A_m$, consisting of small dense blocks of a *fixed size* [3]. When multiplying with such a matrix consisting of small dense blocks the code has to load fewer indices j because only one is needed per block. When multiplying with a dense block, elements of x and y can be loaded once and reused several times.

In our approach we store at least two matrices: one contains the small dense blocks of equal size and the other contains the remaining non-zero elements. In [4] another approach is presented: the authors store the *whole* matrix in small dense blocks, at the expense of having to store some zero entries explicitly.

To store the matrix A_i of small dense blocks we use the same data structure as for the original matrix (SSS format), with the exception that we store a whole block for each coordinate pair (i, j) instead of just one value. We build this data structure using a linear-time greedy algorithm that scans the matrix row by row.

Register blocking can be implemented in several ways:

- *Multiplying matrix-by-matrix*: Multiply each matrix A_i with vector x and sum the results.
- *Multiplying row-by-row*: Multiply with all A_i at the same time, row-by-row. When using this variant one can optionally store the nonzero elements in the same sequence as they are accessed, i. e. store dense blocks of *different* size in each row instead of storing the matrices A_i separately.

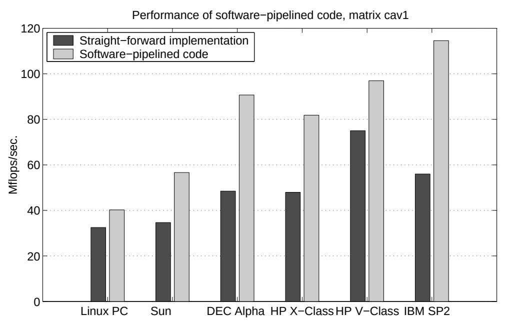
We implemented all above variants. None of them was superior. The optimal routine has to be chosen depending on the matrix and the machine.

2.3 Matrix Reordering

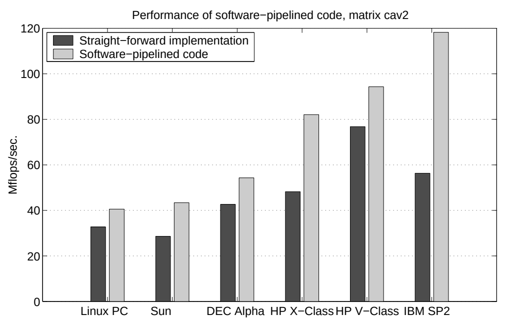
We use Cuthill-McKee *reordering* [5] on the matrix to reduce its bandwidth. Because of the smaller bandwidth it is likely that during matrix-vector multiplication vector elements that are accessed in a particular matrix row will be accessed again in the following row. Thus matrix reordering can reduce cache misses that accesses to x and y generate [3] and more importantly also lowers the number of messages to be sent in the parallel implementation (see section 3).

Name	*cav1*	*cav2*
Size of matrix	17215×17215	54295×54295
# nonzeros	929159	3172021
# nonzeros per row	26.49	28.71
Storage	4.41 MBytes	18.46 MBytes
Properties	symmetric	symmetric

Table 2
Matrices used in numerical experiments. *Both matrices originate from a FEM code that solves Maxwell's Equations in 3D. The amount of dense blocks contained in these matrices are listed in Fig. 2.*



(a)



(b)

Fig. 1. **Performance of the software-pipelined code and the straight-forward code (a) for matrix *cav1* and (b) for matrix *cav2*.** *These experiments are carried out on a single processor.*

2.4 Numerical Experiments

For the numerical experiments we use the matrices listed in Tab. 2. These matrices originate from a FEM code used for the design of particle accelerator cavities, which essentially solves Maxwell's Equations in 3D [6]. The matrices have a large amount of small dense blocks, because three node variables are located at each grid point of the FEM mesh. The exact amount of dense blocks are listed in Fig. 2.

The experiments are carried out on seven different machines as listed in Tab. 1. First we benchmark the three optimisations separately (Figs. 1-3), then we measure the best performance by applying all optimisations at once (Fig. 4).

Fig. 1 shows the performance of software-pipelined code in comparison with the original code from Alg. 1. The benefit is substantial on all platforms. The improvement ranges from 24% on the Intel Linux PC to 110% on the IBM SP2 (for matrix *cav2*).

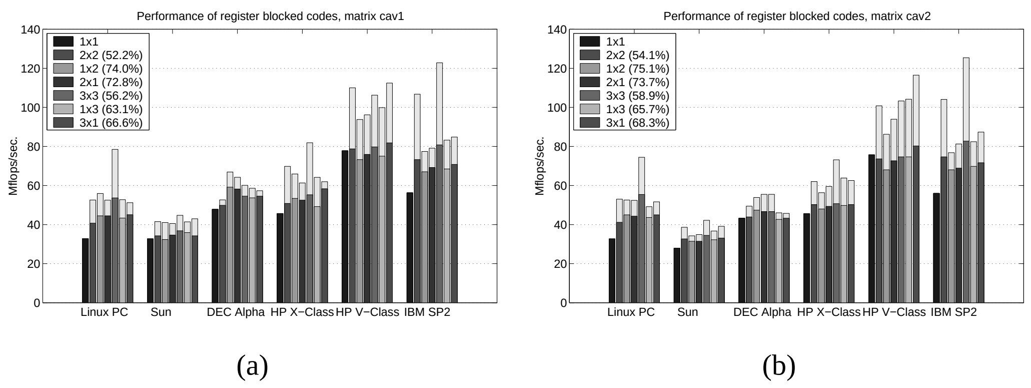


Fig. 2. **Performance of the codes that use register blocking.** *Fig. (a) shows the results for matrix* cav1*, Fig. (b) shows the results for matrix* cav2*. The darker bars (bottom) show the overall performance of the code, including the blocked portion and the unblocked portion. The whole bars (including top parts in light gray) represent the performance of the code portion that multiplies the blocked part of the matrix only. The number in brackets is the percentage of non-zero elements that are stored in dense blocks of the given size. These experiments are carried out on a single processor.*

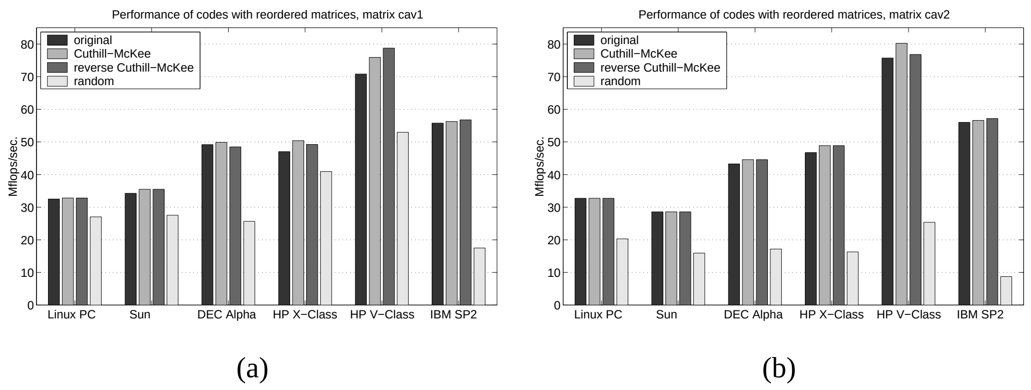


Fig. 3. **Performance of the codes when working on matrices with different orderings.** *Fig. (a) shows the results for matrix* cav1*, Fig. (b) shows the results for matrix* cav2*. These experiments are carried out on a single processor.*

Fig. 2 shows the impact of the block size on the performance of the register-blocked code. For the matrix *cav2* the maximal improvement is 69% on the Intel Linux PC. On the other platforms the improvement lies between 6% and 48%. As can be seen by the lighter colored bars in Fig. 2 the performance of the code can be substantially higher for matrices consisting solely of small dense blocks.

Fig. 3 shows the performance of the unoptimised code when multiplying matrices with different orderings. Compared with the original ordering the performance cannot be increased substantially with Cuthill-McKee-type reorderings. However the experiments with random ordering indicate that the performance depends heavily on the matrix ordering. In cases, where the original ordering is not so well suited for matrix-vector multiplication as in our case, the improvement of Cuthill-McKee-

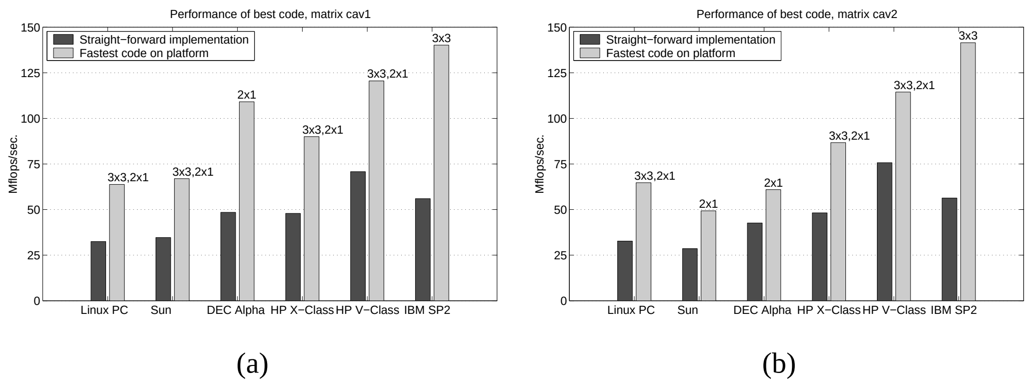


Fig. 4. **Performance of the best code and the unoptimised code.** *Fig. (a) shows the results for matrix* cav1*, Fig. (b) shows the results for matrix* cav2*. For each architecture we use the code that performs best and compare it against the unoptimised code. The labels on top of the bars show the block sizes that yield the best performance for register blocking. (E.g.* "3x3, 2x1" *means that the matrix is splitted into three matrices, one containing all* 3x3*-blocks, the next containing the remaining* 2x1*-blocks and the third containing the remaining elements.)*

type reorderings is more substantial [3].

For each platform we choose the fastest code that takes all discussed optimisations into account and compare it with the corresponding unoptimised version. The results are shown in Fig. 4. On the SP2 we achieve an overall improvement of 151%, on the Intel Linux PC we still get an improvement of 97%, while on the HP X-Class, Sun Enterprise Server, HP V-Class and DEC Alpha Workstation we get performance increases of 80%, 73%, 51% and 43%.

3 Parallelisation using message-passing

3.1 Parallel Implementation

For the parallel implementation we distribute the lower triangular part of the matrix A by block-rows (see Fig. 5). To balance the load we assign the same number of nonzeros to each processor. The distribution of the vectors x and y corresponds to the distribution of the matrix rows.

In a preprocessing step each processor collects the necessary information for the actual matrix-vector multiplication. This is done in the following way:

- The storage format of the matrix implies that processor i needs only elements of the local parts of the x-vector from processor j where $j < i$. For this purpose, the smallest block containing all the needed elements is determined. These

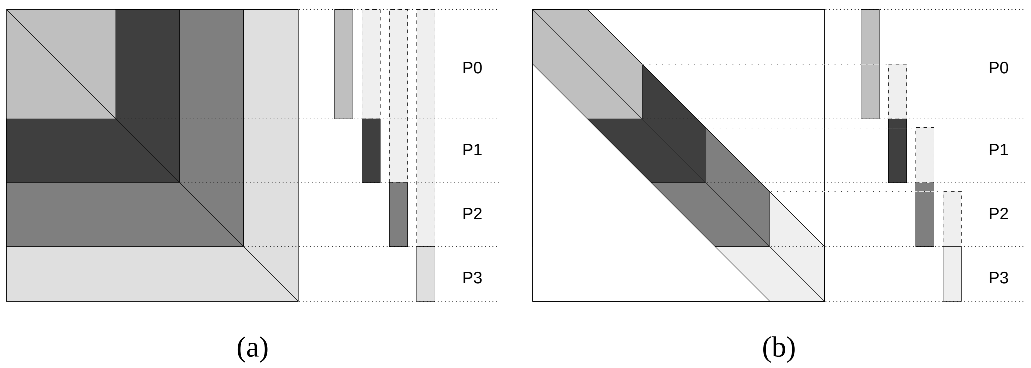


Fig. 5. **Data distribution for parallel implementation.** *The figure shows how the matrix is distributed across the processors for* (a) *a non-banded and* (b) *a banded matrix. Because the matrix is symmetric, only its lower triangle is stored. The vectors depicted on the right show the local parts of the x-vector and which parts of the x-vector must be known to each processor for the multiplication of the local part of the matrix.*

informations are exchanged.

- During the actual matrix-vector multiplication processor i receives the same portion of the y-vector from processor j as it sends to the same processor of the x-vector. This is due to the symmetry of the matrix. As a consequence, no information has to be exchanged for the y-vector in the preprocessing step.

For the actual parallel matrix-vector code we implemented three slightly different routines:

(1) *Without latency-hiding*: Exchange parts of x-vector, then multiply with local part of matrix, then exchange parts of y-vector and form resulting vector.
(2) *With latency hiding*: Exchange parts of x-vector and at the same time multiply with local block-column in the upper triangle. Send the y-vector to the other processors. Upon arrival of the remote parts of the x-vector the local block-row in the lower triangle can be multiplied. Upon arrival of the y-vectors form resulting vector.
(3) *With latency hiding*: Exchange parts of x-vector and at the same time multiply with local diagonal block of the matrix [7]. Upon arrival of the remote parts of the x-vector, multiply with the remaining local part of the matrix, then exchange parts of y-vector and form resulting vector.

Routine 1 is a reasonable choice for machines that do not support latency-hiding. Routine 2 has the disadvantage that it does not exploit the symmetry of the matrix well, since the matrix has to be read from memory twice. Routine 3 has the disadvantage that the diagonal block of the matrix has to be stored separately to make the implementation efficient. This has to be done in a preprocessing step.

The parallel algorithm benefits from the matrix reordering done for the optimisation of the serial code. As can be seen from Fig. 5 the number of messages is reduced because of the smaller matrix bandwidth. Fig. 5 a) shows the worst case: the last

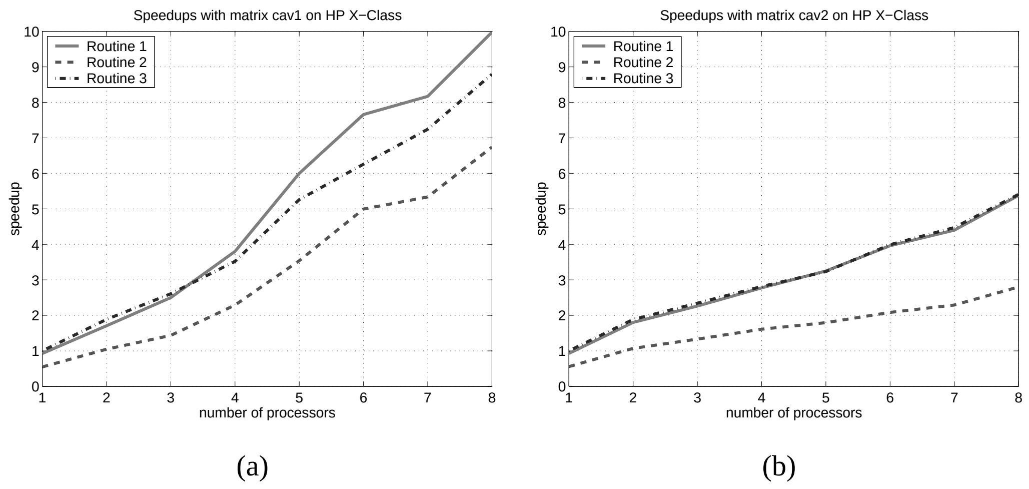


Fig. 6. **Speedups of the parallel matrix-vector multiplication code on the HP X-Class** *Speedups are reported for matrices* (a) cav1 *and* (b) cav2 *using Cuthill-McKee ordering.*

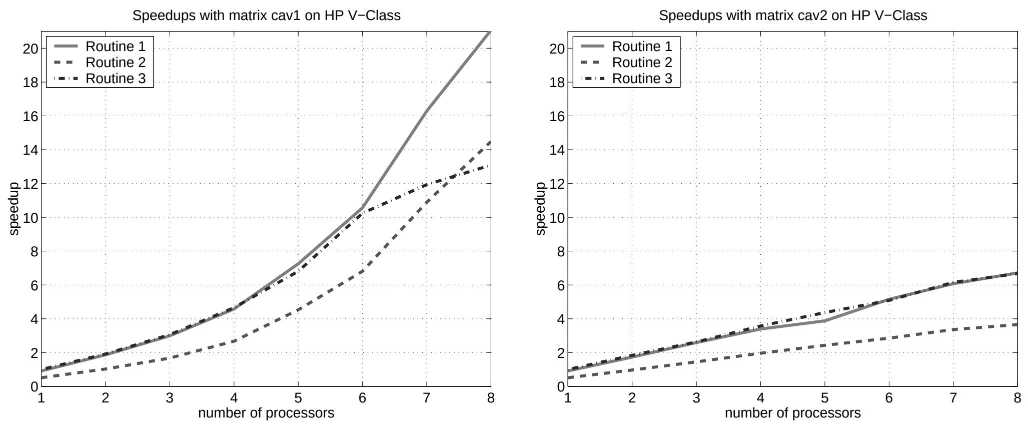
processor (P3) needs the local x-vector parts from all other processors for the multiplication. Whereas in Fig. 5 b) the same processor needs only the local parts of the neighbour processor. The results in Fig. 11 show how crucial the matrix reordering is.

3.2 Parallel Numerical Experiments

We carried out the parallel experiments on five platforms: the HP X-Class, the HP V-Class, the Intel Paragon, the Intel Pentium III Beowulf Cluster and the IBM SP2. The HP X-Class (HP Exemplar SPP2000/X-32) and the HP V-Class (HP Exemplar V2500) are both shared memory machines with 32 processors and a crossbar-switch interconnection network. The Intel Paragon has 128 processors with distributed memory arranged in a 2D-grid. The Intel Beowulf Cluster consists of 251 dual CPU Pentium III processors. These 251 computing nodes are grouped into frames of 24 nodes. An Ethernet network connects the computing nodes. The IBM SP2 is a distributed memory machine with 64 processors connected through a multistage network.

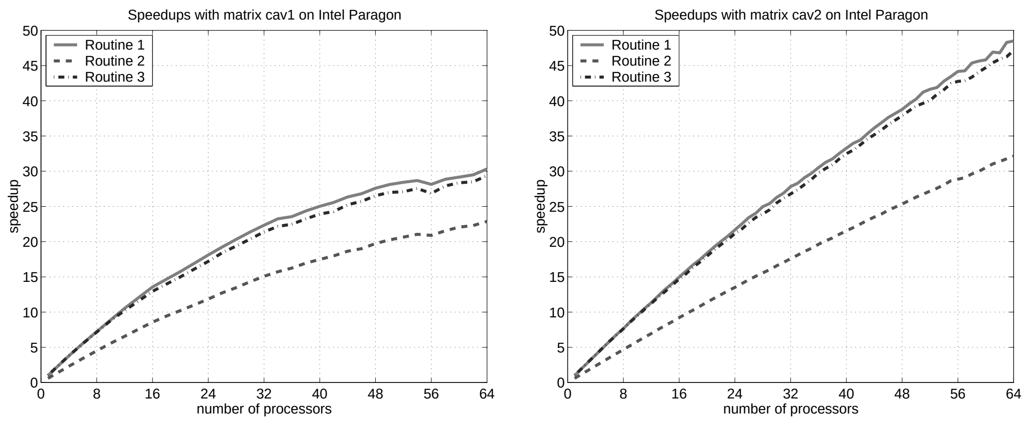
The software-pipelining optimisation described in section 2 was incorporated into the parallel version. Although it would be entirely possible to implement register blocking also for the parallel version, we did not do that, mainly because of the limited time available. Unless otherwise mentioned the matrices are reordered using the Cuthill-McKee algorithm.

Fig. 6 shows the measured speedups for the HP X-Class. For the smaller matrix *cav1* we get super-linear speedup due to cache-effects. The results for the HP V-Class are shown in Fig. 7. On this platform we even get a higher super-



(a) (b)

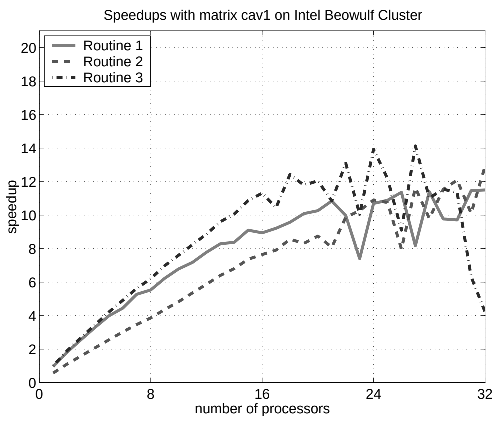
Fig. 7. **Speedups of the parallel matrix-vector multiplication code on the HP V-Class** *Speedups are reported for matrices* (a) cav1 *and* (b) cav2 *using Cuthill-McKee ordering. Note the scaling of the diagrams and the high speedups reached with* cav1.



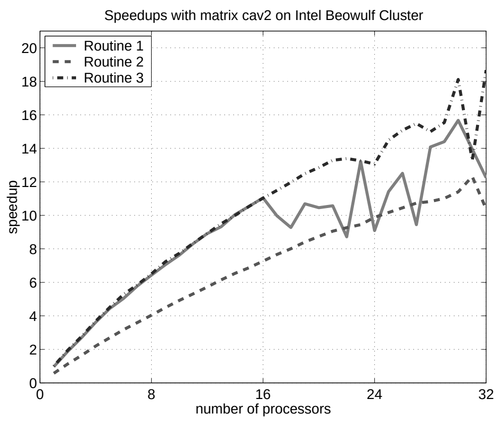
(a) (b)

Fig. 8. **Speedups of the parallel matrix-vector multiplication code on the Intel Paragon.** *The matrices* (a) cav1 *and* (b) cav2 *are reordered using the Cuthill-McKee algorithm.*

linear speedup for matrix *cav1*. For this matrix the speedup is 21 with 8 processors. The speedups for matrix *cav2* are higher compared to the HP X-Class. The speedup for 8 processors is 6.7. Fig. 8 shows the measured speedups for the Intel Paragon. The code scales well, especially for the matrix *cav2*. Apart from the super-linear speedups on the HP X-Class and the HP V-Class, this machine gives the best speedups, because of its fast network compared to the performance of its processors. The results for the Intel Beowulf Cluster are depicted in Fig. 9. For the numerical experiments we used 1 CPU per node. Up to 8 CPU's, the measured speedups for the matrix *cav2* are comparable to the speedups on the HP V-Class. The irregularities above 16 processors show up because in this case some processors have to communicate with processors located in another frame. Fig. 10 shows

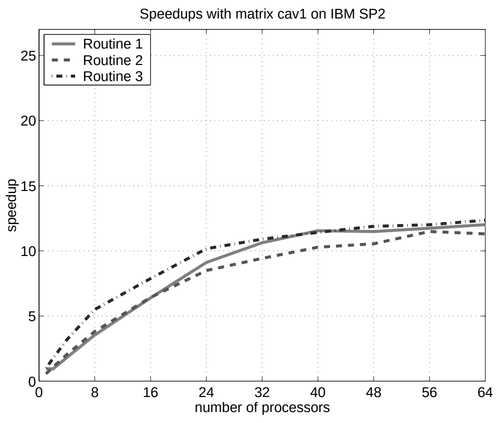


(a)

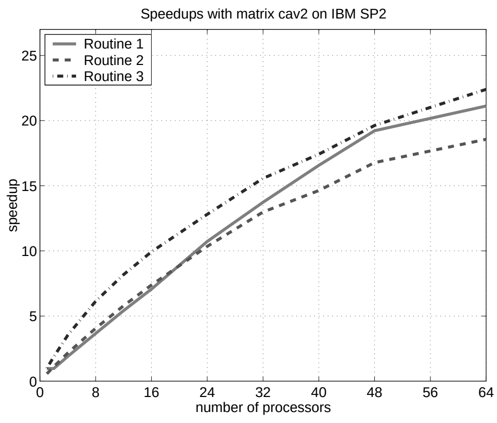


(b)

Fig. 9. **Speedups of the parallel matrix-vector multiplication code on Intel Beowulf Cluster** *Speedups are reported for matrices* (a) cav1 *and* (b) cav2 *using Cuthill-McKee ordering.*



(a)

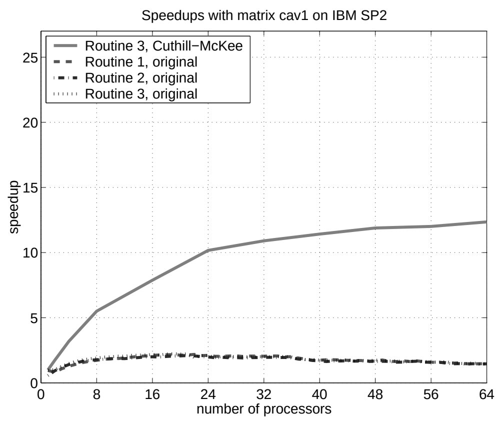


(b)

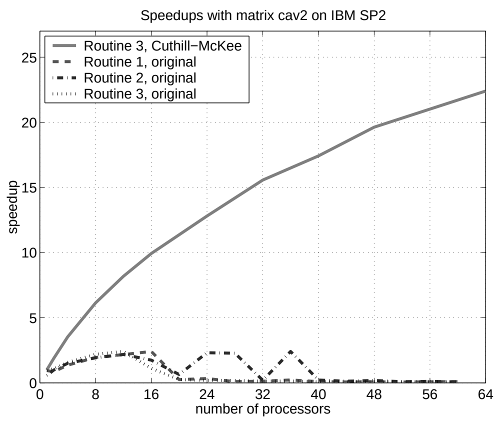
Fig. 10. **Speedups of the parallel matrix-vector multiplication code on the IBM SP2.** *Speedups are reported for matrices* (a) cav1 *and* (b) cav2 *using Cuthill-McKee ordering.*

the measured speedups for the IBM SP2. The code does not scale as well as on the Intel Paragon, because the SP2 has much faster processors and the slower interconnection network.

Fig. 11 shows the influence of the reordering on the performance. When the matrices are left in their original ordering the performance is unacceptably low. Even for a small number of processors, where the number of messages is low, the performance is worse. These results are conducted on the IBM SP2, but this behavior can also be observed on the other platforms.



(a)



(b)

Fig. 11. **Speedups of the parallel matrix-vector multiplication code on the IBM SP2 with and without matrix reordering.** *Speedups are reported for matrices* (a) cav1 *and* (b) cav2 *using their original ordering and Cuthill-McKee ordering. For the reordered matrices only Routine 3 is shown, which gives the best results in this case.*

4 Conclusions

We computed an upper bound for the performance of the sparse matrix-vector product and showed that straight-forward implementations perform poorly. The three techniques we presented in this paper improved the performance by up to 151%. Our message-passing implementation also benefits from these optimisations and scales reasonably. We think that future work should go into automatic generation of sparse matrix-vector multiplication codes which are optimised to a given matrix and a given target architecture. This approach has been successfully applied to other applications, such as the FFT [8] and the dense BLAS [9].

5 Acknowledgments

We would like to thank Peter Arbenz and Oscar Chinellato for improvements and corrections of a draft of this paper. We would also like to thank Rolf Strebel for explanatory discussions on the subject of software pipelining.

References

[1] Y. Saad, SPARSKIT: A basic tool kit for sparse matrix computations, Tech. Rep. 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA (1990).

[2] K. Wadleigh, A. Potler, Advanced optimization for PA-8x00 processors, Presentation at the HiPer'98 conference in Zurich (1998).

[3] S. Toledo, Improving memory-system performance of sparse matrix-vector multiplication, in: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, 1997.

[4] E. Im, K. Yelick, Optimizing sparse matrix-vector multiplication on SMPs, in: Ninth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, March 1999.

[5] A. George, J. W. Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[6] P. Arbenz, R. Geus, A comparison of solvers for large eigenvalue problems occuring in the design of resonant cavities, Numerical Linear Algebra with Applications 6 (1999) 1–13.

[7] Y. Saad, A. Malevsky, P-SPARSLIB: A portable library of distributed memory sparse iterative solvers, Tech. Rep. UMSI 95-180, MSI (1995).

[8] M. Frigo, S. Johnson, FFTW: An adaptive software architecture for the FFT, in: ICASSP, 1998, p. 1381.

[9] R. Whaley, J. Dongarra, Automatically Tuned Linear Algebra Software (ATLAS), in: SC '98 Proceedings, 1998.