Experiments with Data Layouts

M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam

CPDC-TR-97-06

October 1997

Center for Parallel and Distributed Computing

Department of Electrical & Computer Engineering

Northwestern University

Technological Institute

2145 Sheridan Road, Evanston IL-60208

Experiments with Data Layouts

M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam

1 Experimental Results

1.1 Experimental Suite

We present performance results for eight programs: `matmult` is a matrix–multiplication routine with two versions (with and without loop unrolling); `syr2k` is banded matrix update routine from BLAS [10]; `adi` is from Livermore kernels [22]; `btrix`, `vpenta` and `cholesky` are from Spec92/NASA benchmark suite [9]; and `transpose` is a matrix transpose program from NWchem [23], a software package for computational chemistry.

`matmult`, `syr2k` and `transpose` are programs consisting of a single loop nest whereas the others contain multiple loop nests.

For each program in the suite, we experiment with four different versions: `col`– original program with column-major layout for all arrays; `row`– original program with row-major layout for all arrays; `lopt`– a version optimized by loop transformation techniques; and `dopt`– a version optimized by data layout transformations studied in this paper. `lopt` versions are obtained by either applying Li's locality optimization approach [18] to the program, or allowing the native compiler to derive the best order.

In the experiments on multiple nodes, where possible, the coarsest granularity of parallelism is exploited for all versions. In fact, for `row`, `col` and `dopt` versions the same loops are parallelized; therefore the degree of parallelism is the same. The degree of parallelism for `lopt` is either the same as or better than the other versions.

1.2 Platforms

We report execution times in seconds up to 8 processors in Convex Exemplar and SGI Origin, two widely used distributed–shared–memory machines.

Convex Exemplar uses 180MHz HP PA-RISC 8000 processors with 1MB data and 1MB instruction cache. The Exemplar is built around the hypernode configuration. Each hypernode can have up to 16 processors, 16GB of memory and a CTI cache of size 256MB. The programs are compiled by the native C compiler with -O3 option. The timings are taken using the CXpa performance analyzer.

Origin uses 195MHz R10000 processors, 32KB L1 data cache and 4MB L2 unified cache. The programs are compiled by the native C compiler using -O2 option. The timings are taken using *gettimeofday* routine.

1.3 Performances

In this subsection we present miss rates (Table 1) for our suite in uniprocessor as well as the execution times in Exemplar and Origin.

1.3.1 Matrix-Multiplication without unrolling

The code for matrix-multiplication (matmult) is as follows:

```
real C(N,N), A(N,N), B(N,N)

  do i = 1, N
    do j = 1, N
      do k = 1, N
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
      enddo
    enddo
  enddo
```

Wei Li's approach [18] optimizes locality by permuting the loops from ijk to jki, assuming a column-major layout. Our framework, on the other hand, improves spatial locality by assigning row-major layout to A and C, and column-major layout for B.

Table 1 shows the miss rates for uniprocessor case with different cache and line sizes. In the simulation, 256×256 matrices are used. We first note that `row` is better than `col`. This is because `col` exploits spatial locality for array B in the innermost loop, and for array A in the outermost loop whereas `row` exploits spatial locality for array A in the innermost loop and for array B in the second innermost loop. We note that miss rates for `lopt` and `dopt` are very close to each other, `dopt` being slightly better.

Figures 1(a) and 2(a) shows the execution times for those versions in Exemplar and Origin with 1024×1024 double arrays. In all versions, only the outermost loop is parallelized. Figure 1(a) indicates that the performance of `dopt` is much better than those of `row` and `col`. The `dopt` version is also better than `lopt` beyond 2 processors. In fact, as far as the innermost loop is concerned both `lopt` and `lopt` exploit spatial locality for two references and temporal locality for one reference. The situation in Origin (Figure 2(a)) is slightly different. Although, as in Exemplar, `dopt` performs much better than `col` and `row`, except for uniprocessor case `lopt` outperforms `dopt`.

1.3.2 Matrix-Multiplication with unrolling

Table 1 shows the miss rates for matrix multiplication when the outermost loop is unrolled by a factor of 4 and jammed (denoted by `matmult/u` in the table). The results show that `dopt` is always better than the remaining versions. We run unrolled matrix–multiplication code in Exemplar and Origin. In the Exemplar, two outermost loops are unrolled by a factor of 4 and jammed. The cleaning-up code is eliminated. As shown

Table 1: Uniprocessor miss rates for the programs in our experiment suite [`bs` = block (line) and `cs` = cache size].

Program	bs	cs = 8K				cs = 32K				cs = 128K				cs = 512K			
		col	row	lopt	dopt	col	row	lopt	dopt	col	row	lopt	dopt	col	row	lopt	dopt
matmult	16	0.441	0.317	0.226	0.223	0.394	0.271	0.156	0.154	0.383	0.259	0.138	0.136	0.012	0.011	0.013	0.011
matmult	32	0.382	0.320	0.182	0.179	0.336	0.274	0.100	0.098	0.324	0.262	0.080	0.078	0.014	0.012	0.014	0.012
matmult	64	0.357	0.327	0.165	0.162	0.312	0.281	0.078	0.076	0.300	0.270	0.056	0.054	0.020	0.019	0.021	0.019
matmult	128	0.354	0.339	0.167	0.164	0.310	0.296	0.078	0.076	0.299	0.285	0.056	0.054	0.033	0.034	0.035	0.033
matmult/u	16	0.255	0.317	0.287	0.177	0.185	0.130	0.101	0.072	0.167	0.084	0.054	0.045	0.013	0.014	0.016	0.012
matmult/u	32	0.245	0.321	0.274	0.158	0.175	0.133	0.088	0.058	0.158	0.087	0.042	0.033	0.016	0.015	0.017	0.013
matmult/u	64	0.219	0.327	0.274	0.155	0.144	0.141	0.071	0.057	0.125	0.094	0.041	0.032	0.022	0.022	0.023	0.019
matmult/u	128	0.217	0.341	0.283	0.166	0.141	0.155	0.099	0.068	0.121	0.109	0.052	0.043	0.035	0.037	0.038	0.034
syr2k	16	0.664	0.417	0.365	0.206	0.658	0.291	0.316	0.179	0.527	0.173	0.281	0.157	0.100	0.041	0.131	0.013
syr2k	32	0.662	0.542	0.187	0.109	0.655	0.353	0.160	0.092	0.514	0.105	0.142	0.080	0.058	0.024	0.067	0.007
syr2k	64	0.660	0.628	0.101	0.066	0.650	0.416	0.082	0.049	0.501	0.091	0.073	0.043	0.044	0.019	0.034	0.004
syr2k	128	0.659	0.646	0.072	0.057	0.640	0.525	0.043	0.030	0.492	0.246	0.038	0.025	0.051	0.023	0.018	0.003
vpenta	16	0.652	0.502	0.502	0.457	0.652	0.498	0.498	0.457	0.435	0.403	0.403	0.311	0.218	0.272	0.272	0.207
vpenta	32	0.543	0.321	0.321	0.251	0.544	0.317	0.317	0.250	0.475	0.294	0.294	0.177	0.133	0.166	0.166	0.131
vpenta	64	0.489	0.236	0.236	0.151	0.489	0.226	0.226	0.148	0.487	0.217	0.217	0.111	0.236	0.113	0.113	0.072
vpenta	128	0.462	0.227	0.227	0.139	0.462	0.181	0.181	0.098	0.487	0.177	0.177	0.078	0.433	0.087	0.087	0.043
cholesky	16	0.615	0.730	0.206	0.352	0.481	0.575	0.183	0.285	0.263	0.326	0.148	0.192	0.167	0.189	0.082	0.133
cholesky	32	0.566	0.706	0.158	0.177	0.484	0.658	0.104	0.143	0.340	0.372	0.074	0.096	0.114	0.168	0.041	0.066
cholesky	64	0.0561	0.659	0.169	0.089	0.529	0.658	0.098	0.072	0.362	0.411	0.038	0.048	0.144	0.200	0.021	0.033
cholesky	128	0.411	0.593	0.190	0.045	0.349	0.590	0.096	0.036	0.237	0.445	0.020	0.024	0.099	0.304	0.011	0.017
btrix	16	0.228	0.221	0.223	0.223	0.196	0.180	0.196	0.182	0.137	0.099	0.137	0.120	0.118	0.075	0.118	0.091
btrix	32	0.142	0.152	0.142	0.135	0.117	0.109	0.117	0.100	0.079	0.053	0.079	0.064	0.066	0.039	0.066	0.048
btrix	64	0.108	0.139	0.103	0.099	0.075	0.085	0.075	0.061	0.049	0.040	0.049	0.036	0.039	0.020	0.039	0.026
btrix	128	0.118	0.243	0.110	0.110	0.054	0.096	0.054	0.043	0.033	0.040	0.033	0.022	0.024	0.017	0.024	0.014
adi	16	0.493	0.466	0.350	0.388	0.465	0.349	0.234	0.267	0.310	0.310	0.156	0.178	0.282	0.301	0.156	0.162
adi	32	0.375	0.479	0.319	0.343	0.312	0.330	0.119	0.135	0.202	0.286	0.079	0.090	0.182	0.280	0.078	0.082
adi	64	0.272	0.475	0.322	0.346	0.160	0.367	0.132	0.134	0.102	0.306	0.040	0.047	0.091	0.302	0.039	0.043
adi	128	0.356	0.555	0.412	0.452	0.186	0.465	0.249	0.278	0.053	0.311	0.022	0.025	0.046	0.307	0.020	0.023
transpose	16	0.625	0.625	0.625	0.501	0.375	0.375	0.375	0.250	0.281	0.281	0.281	0.250	0.258	0.258	0.258	0.250
transpose	32	0.813	0.813	0.813	0.753	0.344	0.344	0.344	0.125	0.180	0.180	0.180	0.125	0.139	0.139	0.139	0.125
transpose	64	0.906	0.906	0.906	0.874	0.453	0.453	0.453	0.063	0.130	0.130	0.130	0.063	0.080	0.080	0.080	0.063
transpose	128	0.953	0.953	0.953	0.938	0.445	0.445	0.445	0.031	0.104	0.104	0.104	0.031	0.050	0.050	0.050	0.031

in Figure 1(b), except for the uniprocessor case, `dopt` is always better. In the Origin, on the other hand, (see Figure 2(b)) only the outermost loop is unrolled (by 4) and jammed. The `lopt` version outperforms `dopt` for all processor sizes, though the performances are very similar. The matrix multiplication code is an example of a group of programs where loop transformations and data transformations generate competitive codes from the locality point of view.

1.3.3 syr2k

The `syr2k` program (from BLAS [10]) is a loop nest which performs banded rank-2k update. We used `syr2k` to show that our data layout transformation framework can handle complicated access patterns as well. The original code is as follows:

```
real C(N,2b-1), A(N,2b-1), B(N,2b-1)

  do i = 1, N
    do j = i, min(i+2*b-2,N)
      do k = max(i-b+1,j-b+1,1), min(i+b-1,j+b-1,N)
```

```
      C(i,j-i+1) = C(i,j-i+1) + A(k,i-k+b)*B(k,j-k+b) + B(k,i-k+b)*A(k,j-k+b)
    enddo
  enddo
enddo
```

Assuming a column-major layout, all the arrays have poor locality. Our layout optimization algorithm detects that array C should be row-major, and arrays A and B should be stored along anti-diagonal. We note that although arrays A and B are referenced twice each, that does not cause any conflict in the layout requirements.

The version that is optimized by loop transformations (from [18], assuming column-major layouts) is as follows:

```
real C(N,2b-1), A(N,2b-1), B(N,2b-1)

  do u = max(2-2b,2-N-b,0), min(-2+2b,-2+N+b,-2+2N,-1+N)
    do v = u+2min(-u+1-b,1-b,1-N,-u+1-N), u+2min(-u-1+b,-1+b,-u-1+N), 2
      do w = max(1,(u+v)/2+1), min(N,(-u+v)/2+N,(u+v)/2+N)
        C((-u-v)/2+w,u+1) = C((-u-v)/2+w,u+1) + A(w,(-u-v)/2+b)*B(w,(u-v)/2+b)
                + B(w,(-u-v)/2+b)*A(w,(u-v)/2+b)
      enddo
    enddo
  enddo
```

Our layout transformed version, on the other hand, is as follows:

```
real C(2b-1,N), A(N,2:N+2b-1), B(N,2:N+2b-1)

  do i = 1, N
    do j = i, min(i+2*b-2,N)
      do k = max(i-b+1,j-b+1,1), min(i+b-1,j+b-1,N)
        C(j-i+1,i) = C(j-i+1,i) + A(k,i+b)*B(k,j+b) + B(k,i+b)*A(k,j+b)
      enddo
    enddo
  enddo
```

Table 1 shows the simulated miss rates for the different versions with 256×256 double arrays in uniprocessor. The results show that `dopt` outperforms the other versions. In some cases, `lopt` performs even worse than `row`. A reason for this is the fact that while optimizing the spatial locality for arrays A and B, the loop optimization converts the temporal locality for array C into spatial locality. This is a side effect of the iteration-space based optimization algorithms causing sometimes for references with temporal locality to perform worse.

In the run-time experiments, only the outermost loop is parallelized. Figure 1(c) shows the execution times for the `syr2k` code with 1024×1024 double arrays in the Exemplar. The results are compatible with the miss rates. The `lopt` version does not perform as well as expected. In fact, beyond uniprocessor, it performs worse than even `row`. The locality conversion problem mentioned above as well as the complicated loop bounds and subscript expressions can be blamed for this poor performance. Our version (`dopt`), however, outperforms the rest for all processor sizes. The results for the Origin are given in Figure 2(c) and are quite similar in nature to those for the Exemplar. The `syr2k` code is an example of a class of programs where data layout optimizations generate better code than loop transformations; although loop transformations are applicable and improve the performance of the original code somewhat.

1.3.4 vpenta

The `vpenta` benchmark is from Spec92/NASA benchmark suite [9]. Table 1 shows the miss rates for four versions of `vpenta` for different cache parameters. The problem size is 720. The spatial locality in `col` version is poor for all references. Our optimization framework stores the two-dimensional arrays row-major, and the three-dimensional arrays in such a way that the second index position will be the fastest changing dimension in memory. We note that the miss rates for `dopt` are lower than those of `lopt`.

We conduct run-time experiments with four versions with an input size of 920 in Exemplar (see Figure 1(d)) and of 720 in Origin (see Figure 2(d)). The program is placed inside a timing loop which iterates 50 times. It should be emphasized that `lopt` version of this benchmark improves parallelism as well; consequently has an additional advantage (which does not reflect on uniprocessor cache miss rates, of course). In particular, a two-deep loop nest in the program is optimized for coarsest granular of parallelism. The data transformations, on the other hand, cannot optimize parallelism. Despite this fact, in Exemplar, `dopt` outperforms `lopt` in 1 and 3 processor cases. In 2 and 4 processor cases `dopt` and `lopt` performs the same. Beyond four processors, all four versions perform similar. The performance results in Origin are quite similar.

The results for this benchmark show that data layout transformations can generate competitive codes with loop transformations even for the programs consisting of multiple loop nests.

1.3.5 cholesky

The `cholesky` benchmark is also from Spec92/NASA [9]. The miss rates given in Table 1 indicate that although `dopt` performs better than `col` and `row` versions, its performance is not as good as that of `lopt`. This program has a number of loop nests. An important characteristic of this code is that the only paralalelizable loops are in the innermost positions. Our framework can optimize spatial locality, but in multiprocessors, parallelizing the innermost loops leads to false sharing, as parallelized loops will also carry spatial reuse. Because of this reason, the performance of `dopt` is poor (although in uniprocessor case, it outperforms `col` and `row` significantly).

It is interesting to note that even an iteration space based locality optimization framework such as proposed in [18] cannot optimize locality and parallelism for this program. However, the commercial compiler in the Exemplar optimizes it using a non-trivial sequence of transformations including loop permutation and loop fission. The result (which we call `lopt` for this example) is a much better optimized code which exploits parallelism as well as locality.

We conduct experiments with four versions with size parameters IDA and NMAT are both set to 2500. The results are shown in Figure 1(e) for Exemplar and in Figure 2(e) for Origin. As expected, `lopt` is the clear winner.

1.3.6 btrix

The `btrix` benchmark (also from Spec92/NASA [9]) accesses four 4D arrays. The miss rates are given in Table 1. The original program is first modified such that the maximum degree of parallelism is obtained. Since the code contains a number of parallel loops with a nest depth of one, all versions incur false sharing, the impact of which is much more severe in Origin and with the `row` and `dopt` versions.

We present the execution times in Figure 1(f) for the Exemplar and in Figure 2(f) for the Origin. The size parameters JD, KD, LD and MD are all set to 150. Our results show that up to 4 processors, `dopt` is quite successful. Beyond 4 processor, however, the false sharing effect dominates and both `row` and `lopt` outperform `dopt`.

1.3.7 adi

The `adi` code is a kernel from Livermore benchmarks [22]. It consists of two loop nests and accesses three 3D and three 1D arrays. Our data layout transformation framework changes the layouts of 3D arrays, exploiting the spatial locality successfully in the innermost loops in both the nests. `lopt` obtains the same effect by re-ordering the loops. The loop transformation additionally causes the references to 1D arrays have temporal locality, and the difference between the performances of `dopt` and `lopt` in miss rates (see Table 1) comes from this optimization.

The execution times are shown in Figure 1(g) for the Exemplar and in Figure 2(g) for the Origin.

This example also implies the necessity of optimizing layouts of 1D arrays. Our current framework works only for multi-dimensional arrays. We are currently working on a method based on Maslov's delinearization scheme [20] to convert 1D arrays into multi-dimensional ones. Once the delinearization is performed, our technique described in this paper can be applied to those arrays as well.

1.3.8 transpose

The `transpose` code is from NWChem [23], a computational chemistry package developed in PNL. It essentially copies transposes of two arrays to each other.

```
real A(N,N), B(N,N), X

  do i = 1, N
    do j = 1, N
      X = A(i,j)
      A(i,j) = B(j,i)
      B(j,i) = X
    enddo
  enddo
```

No loop transformation can improve spatial locality for both A and B. Our data transformation framework, however, assigns row-major layout for A, and column-major layout for B.

We make experiments with N = 2KB, and the program is put within a timing loop with a trip count of 50. The miss rates shown in Table 1 indicate that `dopt` outperforms the remaining versions. The majority of the misses in all versions are due to conflicts, which can be eliminated by a set-associative cache. Figure 1(h) and Figure 2(h) show that `dopt` outperforms the remaining versions for all processor sizes on both machines. The performances of `lopt` and `col` are the same and very close to that of `row`.

The `transpose` code is representative of a class of programs where loop transformations cannot optimize locality due to conflicting access requirements among different arrays.

1.4 Observations

From our experiments, we observe the following:

- The data layout transformations are very effective in optimizing cache locality for uniprocessors. To be precise, five out of eight cases in Convex as well as Origin, the `dopt` version outperforms the others in single node.

- Our transformations can optimize some programs such as `syr2k` and `transpose` which cannot be fully optimized by loop transformations.

- In programs where there are multiple nests, our framework is quite successful in detecting the optimal layouts due to the fact that (except `btrix`) there was no conflict between the optimized layout requirements of individual nests.

- Although in general data layout optimizations can be considered successful for multiprocessor case, in cases where outermost loop parallelism cannot be obtained, the performance may be rather poor.

- The relative performances of different versions across the machines change slightly. We attribute this to the different cache sizes and organizations of the Exemplar and Origin.

References

[1] B. Appelbe, and B. Lakshmanan. Optimizing parallel programs using affinity regions. *Technical Report GIT-ICS-92/59, Georgia Institute of Technology*, November 1992.

[2] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[3] J. M. Anderson, and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.

[4] U. Banerjee. Unimodular transformations of double loops. In *Proc. Advances in Languages and Compilers for Parallel Processing*, edited by A. Nicolau et.al. MIT Press, 1991.

[5] D. Callahan, S. Carr, and K. Kennedy. Improving register allocations for subscripted variables. *Proc. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, La Jolla, California, June 1990, pages 53–65.

[6] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, CA, Oct 1994.

[7] M. Cierniak, and W. Li. Unifying data and control transformations for distributed shared memory machines. *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995.

[8] M. Cierniak, and W. Li. Recovering logical data and code structures. Technical Report 591. Dept. of Computer Science, University of Rochester, July 1995.

[9] K. M. Dixit. New CPU benchmark suites from SPEC. In Proc. COMPCON'92 –37^{th} IEEE Computer Society International Conference, San Francisco, CA, Feb 1992.

[10] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. In *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[11] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations, *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[12] M. D. Hill, and A. J. Smith. Evaluating associativity in CPU caches. In *IEEE Trans. on Computers*, C-38(12):1612–1630, December 1989.

[13] C.-H. Huang, and P. Sadayappan. Communication-free partitioning of nested loops. In *Journal of Parallel and Distributed Computing*, 19:90–102 (1993).

[14] Y.-J. Ju, and H. Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Proc. Fourth International Workshop in Languages and Compilers for Parallel Computing*, pages 344–358, Ithaca, NY, 1992.

[15] M. Kandemir, J. Ramanujam, and A. Choudhary. A compiler algorithm for optimizing locality in loop nests. In *Proc. 11th ACM International Conference on Supercomputing*, pages 269–276, Vienna, Austria, July 1997.

[16] M. Kandemir, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. In *Proc. 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT 97)*, pages 236–247, San Francisco, CA, November 1997.

[17] K. Kennedy, and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proc. Languages and Compilers for Parallel Computing: 6th International Workshop*, pages 301–320, Portland, OR, 1994.

[18] W. Li. Compiling for NUMA parallel machines. Ph.D. dissertation, Cornell University, Ithaca, NY, 1993.

[19] W. Li. Compiler cache optimizations for banded matrix problems. In *Proc. 9th ACM International Conference on Supercomputing*, pages 21–30, Barcelona, Spain, July 1995.

[20] V. Maslov. Delinearization: an efficient way to break multi-loop dependence equations. In *Proc. the SIGPLAN'92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[21] K. McKinley, S. Carr, and C.W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems,* 1996.

[22] F. McMahon: *The Livermore Fortran Kernels: A computer test of the numerical performance range - Technical Report UCRL-53745*. Lawrence Livermore National Laboratory, Livermore, CA, 1986.

[23] NWChem: a computational chemistry package for parallel computers, version 1.1, 1995. *High Performance Computational Chemistry Group*, Pacific Northwest Laboratory, Richland, WA 99352.

[24] M. F. P. O'Boyle, and P. M. W. Knijnenburg. Non-singular data transformations: Definition, validity, applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pages 287–297, Aachen, Germany, 1996.

[25] Perfect Club: *The Perfect Club Benchmarks: Effective performance evaluation of supercomputers*. The International Journal of Supercomputing Applications, Vol. 3, No. 3, pp 5-40, 1989.

[26] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. Supercomputing 92*, pages 214–223, Nov 1992.

[27] J. Ramanujam, and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. In *IEEE Transactions on Parallel and Distributed Systems*, 2(4), pages 472–482, Oct 1991.

[28] A. Schrijver. *Theory of Linear and Integer Programming*, Wiley-Interscience series in Discrete Mathematics and Optimization, John Wiley and Sons, 1986.

[29] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, 29(12):31–37, Dec 1994.

[30] M. Wolf, and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation*, pages 30–44, June 1991.

[31] M. Wolfe. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, CA, 1996.

[32] H. Zima, and B. Chapman. *Supercompilers for parallel and vector computers*, ACM Press, NY, 1991.

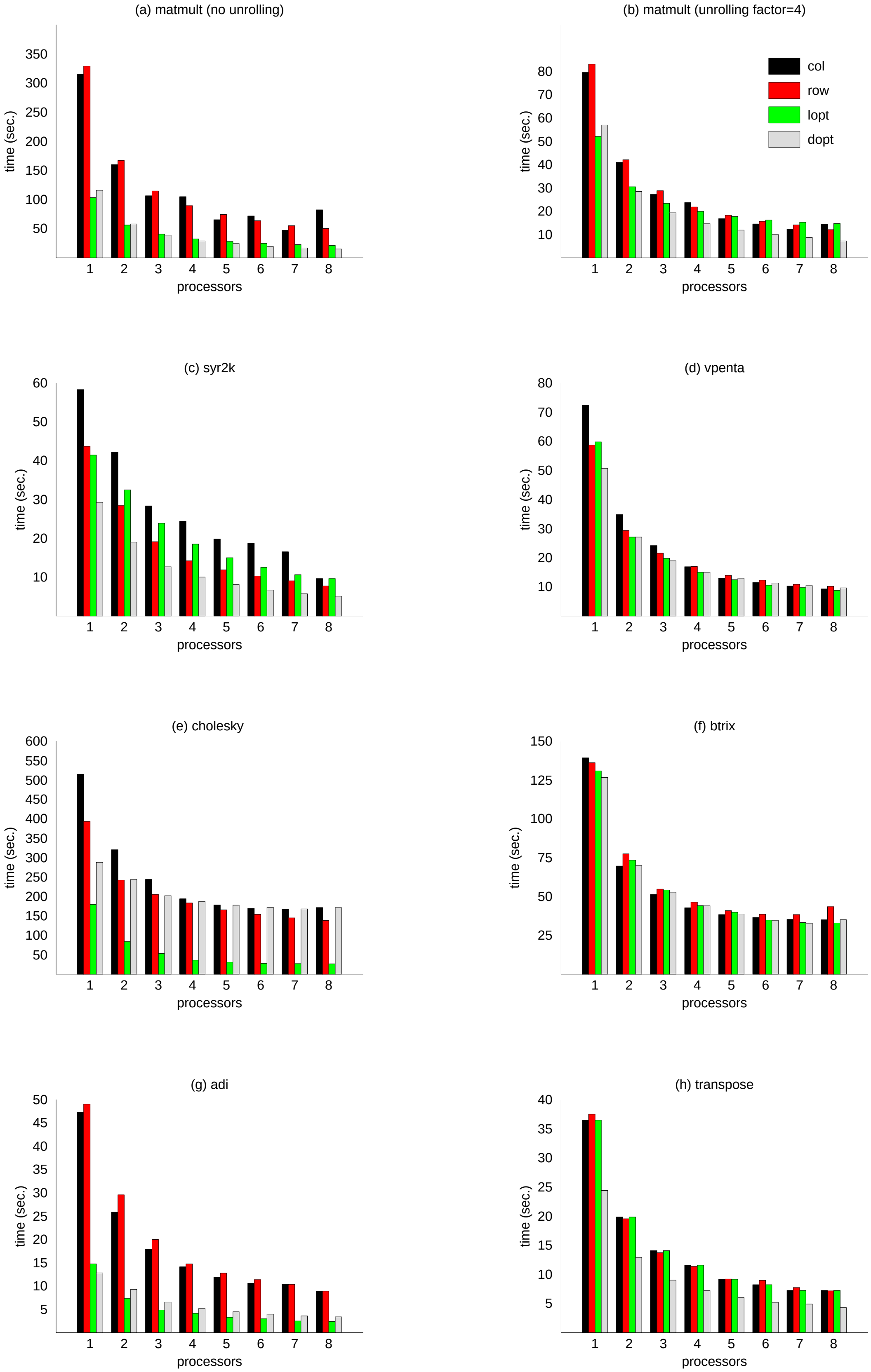

Figure 1: Execution times in Convex Exemplar.

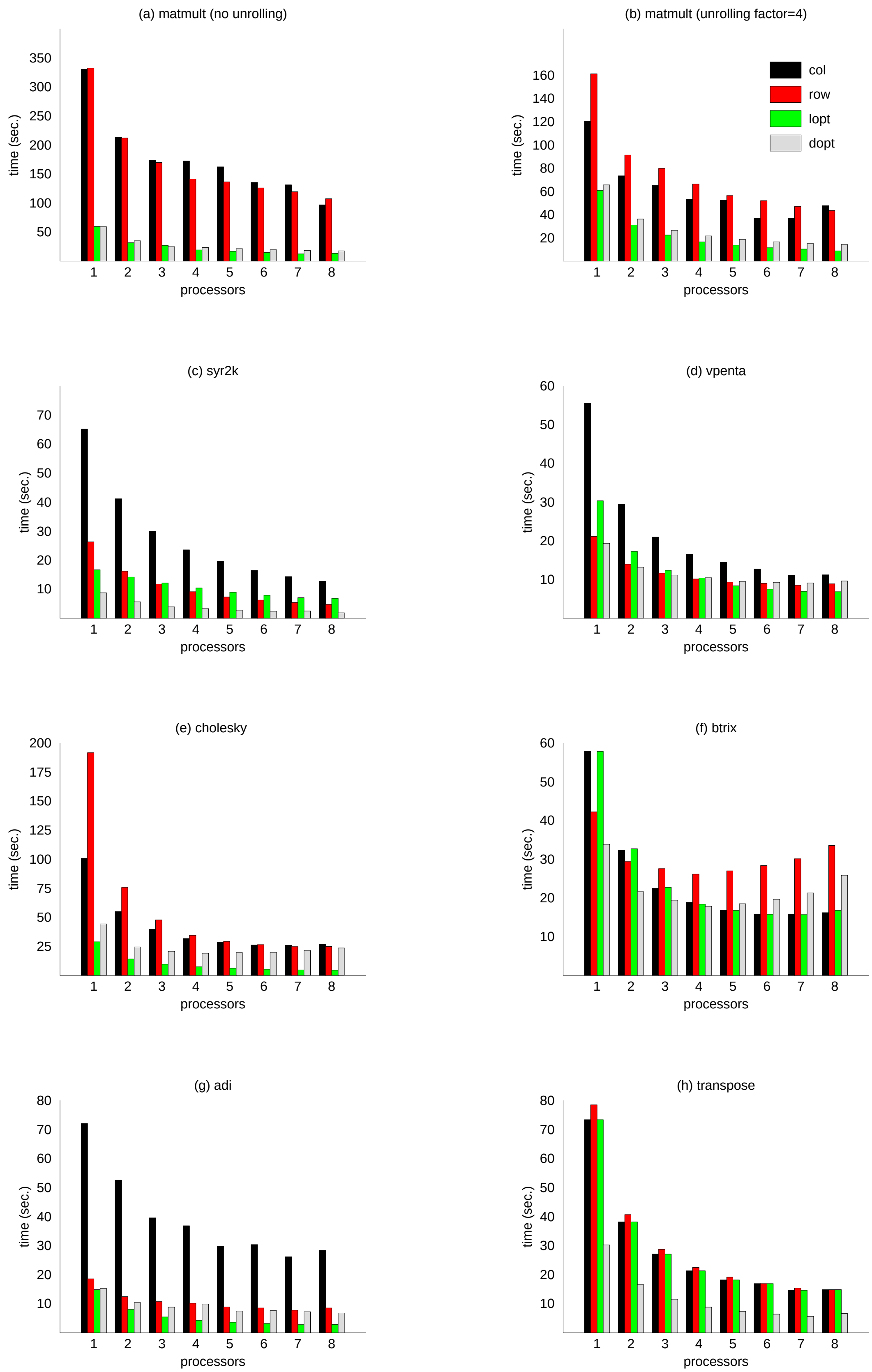

Figure 2: Execution times in SGI Origin.