

Pseudo-Vectorization and RISC Optimization Techniques for the Hitachi SR8000 Architecture

Georg Hager¹, Frank Deserno², and Gerhard Wellein³

Regionales Rechenzentrum Erlangen, Martensstr. 1, 91058 Erlangen, Germany

Summary. We demonstrate optimization techniques for the Hitachi SR8000 architecture on CPU and SMP level using selected case studies from real-world codes. Special emphasis is given to a comparison with performance characteristics of other modern RISC and vector CPUs.

1 Introduction

Vector supercomputers have been carrying the performance crown in numerical applications for over two decades. Superior memory bandwidth, data parallelism and pipelining abilities have made these systems the premier choice for top notch research. With the advent of powerful superscalar RISC processors in the mid-80s, a change in programming style was required that could not be easily implemented in all numerical codes, so that vector architectures were still the only alternative for many applications. The Hitachi SR8000 system, however, is the first parallel RISC system that has competitive memory bandwidth and internal parallelism, and thus provides a flexible “hybrid” environment with both vector and RISC features. It is generally expected that the hybrid approach will be adopted by more and more supercomputer vendors so that vector systems will continually decrease in significance.

There is a price to be paid for this flexibility, though. Hybrid supercomputer architectures like the SR8000 require a thorough insight into inner workings, compiler abilities and ‘weak spots’ in order to exploit their unique features in an optimal way. Using selected examples from real-world codes we illustrate SR8000-specific optimization techniques and compare the results with other architectures.

This section gives a coarse overview to the different benchmark platforms used and some of their peculiarities. The second section discusses basic performance issues with vector triads and sparse matrix-vector multiplication on single CPU and node level. In the third section a RISC optimization strategy for loop kernels of a nuclear physics code is given where careful data flow and

¹ georg.hager@rrze.uni-erlangen.de

² frank.deserno@rrze.uni-erlangen.de

³ gerhard.wellein@rrze.uni-erlangen.de

Table 1. Benchmark platforms and single processor/node specifications. Peak performance numbers (Peak) are given in GFlops, whereas the memory bandwidth (Memb) per processor is given in GBytes/s. For the RISC based systems the L1 and L2 cache sizes are contained

Platform	Single CPU/node specifications				Remarks
	Peak	Memb	L1 [kB]	L2 [MB]	
Intel-P4 1.5 GHz	1.5	3.2	8	0.256	RD-RAM / pgf90
SGI Origin3400	1.0	1.6	32	8	28 way - ccNUMA
HITACHI SR8000-F1 (1 CPU)	1.5	4.0	128	—	PVP
NEC SX5e (1 CPU)	4.0	32.0	—	—	—
HITACHI SR8000-F1 (1 node)	12.0	32.0	1024	—	PVP+COMPAS

locality analysis is important. The fourth section deals with the implementation and shared memory (SMP) parallelization of a strongly implicit solver which is used in many codes from computational fluid dynamics (CFD).

Benchmark platforms The benchmarks have been performed on the platforms given in Table 1. The Hitachi SR8000 processor is basically an IBM PowerPC design running at 375 MHz with important modifications in hardware and instruction set [2]. These extensions include 160 floating point registers and the possibility of up to 16 outstanding `prefetch` or 128 outstanding `preload` instructions. While the `prefetch` loads one cache line in advance to the L1 cache, `preload` instructions can load single data items directly from the memory to the register set, bypassing the L1 cache. Together with an extensive software pipelining done by the compiler these features are called *Pseudo Vector Processing* (PVP), as they are able to provide a continuous stream of data from main memory to the processor avoiding the penalties of memory latency. Considering the programming style, in particular programs with long inner vectorizable loops benefit from PVP. The performance of programs with high cache locality, however, typically suffers from PVP, especially if `preload` instructions are generated for data already resident in L1 cache. The Hitachi SR8000-F1 node comprises eight compute processors accessing a shared memory. Parallelization within the node can be done either by assigning one MPI process to each processor (MPP mode) or by running shared memory parallelization with eight threads (COMPAS mode). Special emphasis has been put by Hitachi on this **C**ooperative **M**icro**P**rocessors in a single **A**ddress **S**pace mode which is a collection of some unique features:

- Sophisticated auto-parallelization compiler features
- Hardware support for fast collective thread operations

- Memory bandwidth of 32 GBytes/s which matches the aggregated single processor bandwidths (note that one SX5e CPU also has a bandwidth of 32 GBtyes/s).
- 512-way interleaved memory to avoid memory contention and to hide bank busy times

Therefore one Hitachi node can be seen either as an eight-way RISC based SMP node or as a vector-like CPU running one process with eight threads. For all our benchmark tests on the SR8000, the compiler option `-oss` was used. For the single-threaded codes, `-noperallel` was specified additionally.

The NEC SX5e processor runs at a frequency of 250 MHz using eight-track vector pipelines. There is one store and two load pipelines, but loads and stores cannot be performed in the same cycle. Thus effectively, the CPU has one load/store pipe which is twice as fast for loads than for stores.

The MIPS R14000 CPU used in the Origin 3400 implements the MIPS-IV instruction set and runs at 500 MHz. This four-way superscalar processor has the ability to sustain four outstanding memory references, including cache line prefetch from main memory to L2 cache. There are no provisions for preload, though. In the Origin 3000 ccNUMA architecture, the four processors of a node share local memory with an aggregated bandwidth of 3.2 GBytes/s, where the maximum for one CPU is 1.6 Gbytes/s. Links between the nodes can transfer data with up to 1.6 GBytes/s in each direction.

Intel's Pentium 4 is the latest incarnation of the IA32 architecture. The single processor system used for benchmarking was equipped with dual-channel RDRAM memory that has a theoretical bandwidth of 3.2 Gbytes/s. The processor has a hardware prefetch mechanism that tries to detect access patterns and prefetches cachelines into L2. As in the MIPS processor there is no `preload` instruction. No use was made of the new SSE2 features of the Pentium 4 processor due to the lack of appropriate compilers.

2 Basic loop kernels

As a starting point two basic loop kernels — the vector-triad and a sparse matrix-vector multiplication — are discussed. In principle, they represent the performance characteristics of the large class of scientific applications with performance mainly bounded by the quality of the memory access, e.g. most engineering applications. Of course, for operations that use data from the different cache levels, the cache bandwidths (latencies) are decisive, which are usually much higher (lower) than the corresponding numbers for the main memory.

2.1 Vector-triad operation

The vector-triad operation $A(1 : N) = B(1 : N) + C(1 : N) * D(1 : N)$ is widely regarded as the most important operation for engineering appli-

cations [3]. With a memory intensity of two memory references per floating point operation (Flop), most processors can not achieve their peak performance even for data already resident in the L1 cache. E.g., the Hitachi processor has a L1-register bandwidth of 32 Bytes (4 Words) per processor cycle and thus can only saturate one of the two multiply-add pipelines, limiting the maximum performance for the vector-triad to 750 MFlops (half of peak performance). Considering that either four `load` or only two `store` operations to the L1 cache can be done, the maximum achievable performance is reduced to 600 MFlops. Measuring values of up to 560 MFlops for intermediate loop lengths ($N \approx 1000 - 5000$) we come quite close to this maximum value as can be seen from Fig. 1. At longer loop length a drop in performance occurs

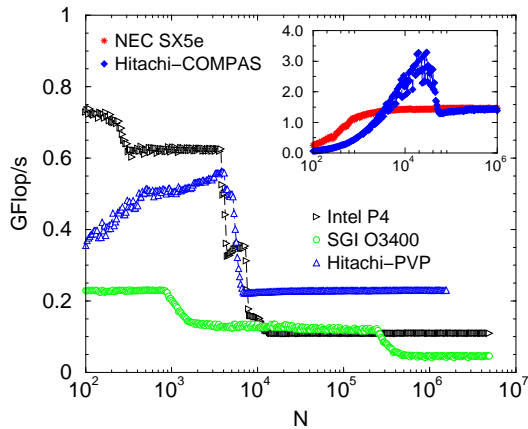


Fig. 1. Performance of vector-triad operation: The main panel depicts cache based systems (PVP enabled for Hitachi SR8000-F1). In the inset performance of one Hitachi SR8000-F1 node (PVP+COMPAS mode) and one NEC SX5e vector processor is shown.

when data has to be loaded from main memory. Since the memory bus can do either two `load` or two `store` operations per memory bus cycle (4 ns), basically two (memory bus) cycles are required to perform two Flops for the vector-triad operation. Therefore — ignoring the memory latency — an upper performance limit of 250 MFlops exists for data resident in main memory. Avoiding the penalties from memory latency by using PVP, more than 90 % of this theoretical limit (230 MFlops) can be maintained for long loop lengths (see Fig. 1). The importance of hiding memory latencies even for simple kernel loops like the vector-triads becomes more visible when comparing Intel and Hitachi performance numbers at long loop lengths: Although the Intel system offers 80 % of the Hitachi memory bandwidth it only achieves half of the performance. Of course at short loop length the Intel system outperforms the Hitachi due to the higher clock speed of the processor. As an example for modern RISC processors with large external L2 caches the SGI numbers are given in Fig. 1 for comparison. Please note that the SGI in-cache performance is mainly limited by the fact that the MIPS processor can only issue

one memory instruction per cycle which gives an effective cache bandwidth of 4 GByte/s for the vector-triads.

An even more interesting discussion is provided by the inset of Fig. 1 where the performance numbers of combined PVP and COMPAS mode using one Hitachi node and of one NEC vector processor are given. Since the memory bandwidth is the same for both systems, we find the same asymptotic performance of about 1.5 GFlops. This value is about two times CRAY T90 performance and is only slightly below the NEC measurements given in Ref. [3]. At low loop lengths the classical vector processor benefits from fast vector start-up times (when compared to the thread synchronization times of the Hitachi node). However, at intermediate loop lengths the Hitachi outperforms the NEC system by more than a factor of two due to the high aggregate L1 cache bandwidth of the eight processors of the Hitachi node.

2.2 Sparse matrix-vector multiplication

Numerical cores of many scientific applications, e.g. quantum mechanical many body problems in theoretical physics, are iterative sparse matrix algorithms such as sparse eigenvalue solvers. In practice, the performance of such operations is mainly determined by a matrix-vector multiplication (MVM) involving a large sparse matrix. Exploiting the sparsity of the matrix two storage formats are widely used to store the nonzero matrix entries only: The *Compressed Row Storage* (CRS) format and the *Jagged Diagonal Storage* (JDS) scheme [5]. Since the JDS format achieves high performance on vector processors and provides only minor drawbacks on RISC based systems a JDS based implementation has been chosen in what follows [6]:

```

for j= 1,...,max_nz do
  for i= 1,...,jd_ptr(j+1)-jd_ptr(j) do
    y(i) = y(i)+ value(jd_ptr(j)+i-1)* x(col_ind(jd_ptr(j)+i-1))
  end for
end for

```

The MVM is performed along the *Jagged Diagonals*, providing an inner loop length essentially equal to the matrix dimension (D_{Mat}), while the outer loop runs over the maximum number of nonzero entries `max_nz` per row. A vectorization and/or automatic parallelization of the inner loop is generally done automatically by the compilers. For a detailed description of our JDS implementation we refer to Ref. [4]. Since the JDS MVM involves a vector gather operation on vector \mathbf{x} it is expected that cache based systems show poor performance numbers for intermediate and large matrix dimensions, while vector processors can sustain a substantial fraction of their peak performance.

Basic performance characteristics of the systems under consideration are presented in Fig. 2, using a benchmark problem from theoretical physics ($D_{\text{Mat}} \approx 10^3 - 10^7$; `max_nz` ≈ 20) [4]. For the cache-based SGI Origin and Intel systems the characteristic performance drop at small/intermediate loop

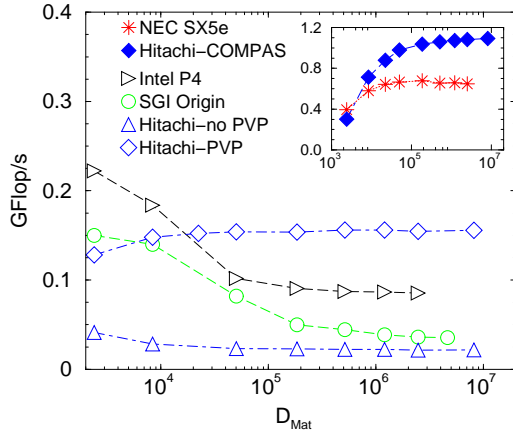


Fig. 2. Performance of sparse MVM kernel: A comparison of cache based systems and single CPU Hitachi SR8000-F1 (PVP enabled/disabled) is depicted in the main panel. The inset shows the performance of one Hitachi SR8000-F1 node (PVP+COMPAS mode) and one NEC SX5e vector processor.

lengths (D_{Mat}) is prominent. Only minor cache effects along with poor performance numbers are obtained for the Hitachi when disabling the PVP feature. However, the performance characteristics change completely if PVP is enabled: The asymptotic performance grows by a factor of nearly 5 to approximately 155 MFlops. Moreover the characteristic performance drop of cache based systems is replaced by a vector processor like behavior, where performance increases with problem size (see also NEC data in inset of Fig. 2) and saturates at a high level. Thus, the memory latency even for the vector-gather operation on vector x can be efficiently hidden by the PVP feature which generates a software pipeline of `prefetch` and `preload` instructions (cf. Fig. 3). The `prefetch` transfers a full cache line (16 or 32 data items depend-

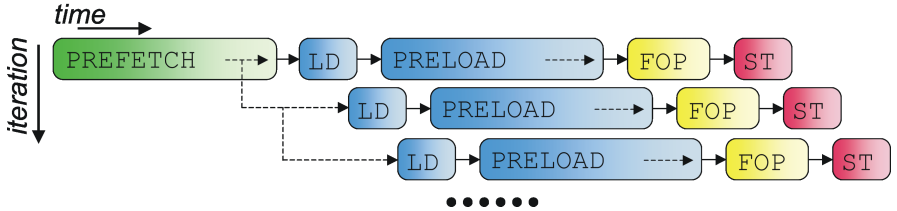


Fig. 3. Diagrammatic view of Pseudo Vector Processing for indirect memory access

ing on the data type) of the index array (`col_ind`) to the L1 cache followed by load instructions. For each entry of the cache line non-blocking `preload` instructions are issued. After paying only once the penalty of memory latency, a continuous stream of data from memory to registers is established for non-contiguous access to vector x . To abide the memory flow in the inner loop of the JDS MVM, the compiler performs loop unrolling at a high level (48 times) and builds a long software pipeline (27 stages). Of course, inter-

mediate to long loop lengths are required to make full use of this technique ($D_{\text{Mat}} \gtrsim 10^4$).

Finally, a surprising result is depicted in the inset of Fig. 2: The performance of one Hitachi node using PVP and COMPAS significantly outperforms the NEC vector processor at intermediate to long loop lengths! Since the asymptotic performance of both systems is the same for the vector triads, this difference can not be attributed to the memory bandwidth but is mainly determined by the implementation of the vector-gather operation. Obviously the combination of PVP and COMPAS can surpass the hardware implementation of the NEC system, if the loop is long enough. Moreover the single processor performance of combined PVP and COMPAS mode (138 MFlops) shows a decrease of less than 10% when compared to the single processor performance, indicating only minor memory contention problems in the COMPAS mode. However, at small problem sizes the vector processor still outperforms the Hitachi node because of short vector start-up times.

For large scale problems a distributed memory parallelization (using MPI) of the JDS MVM is required. In this context, the Hitachi architecture offers a wide variety of programming approaches (pure MPI, MPI+COMPAS, MPI+OpenMP), which have been discussed in detail in Ref. [4].

3 RISC Optimization

In this section we illustrate special RISC optimizations that are especially suited for the Hitachi CPU. The original code is being used for calculations of scattering problems with three-nucleon forces in nuclear physics [7].

For the benchmark considered, the single subroutine MAT took 98% of the total runtime. The routine achieved only about 26 MFlops, and this could be attributed exclusively to the following two loops:

```

DO 40 M=1, IQM                                1
  DO 30 K=KZHX(M), KZAHL                      2
    F(K)=F(K)*S(MVK(K,M))                    3
30  CONTINUE
40  DO 50 K=1, KZAHL                          5
    WERTT(KVK(K)) = WERTT(KVK(K)) + F(K)
50  CONTINUE                                  7

```

Arrays F(), S() and WERTT() are double precision, all other data is integer. For any further optimization work one has to consider the following facts:

- S() is short (about 100–200).
- WERTT() is very short (length 1 in the worst case).
- IQM is very small (typically 9).
- The index array KZHX() contains indices in ascending order.

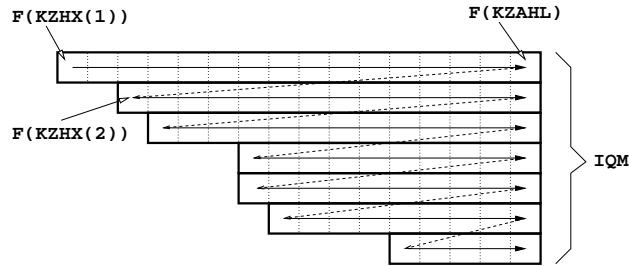


Fig. 4. Original access pattern for $F()$ in the first loop. Outer loop unrolling is impossible.

– $KZAHL$ is typically much larger than 1000.

Performance numbers in MFlops are valid for the whole subroutine, unless otherwise noted.

3.1 First Loop

The Hitachi compiler generates `preload` streams for $S()$ and for $WERTT()$ and `prefetch` streams for $F()$ and $MVK()$. As $S()$ contains only between 100 and 200 elements and $WERTT()$ is even shorter (only one element in the worst case), PVP is just overhead for those two arrays. Thus the first optimization is to switch off any vectorization for $S()$ by `*voption nopreload` directives. This already causes a speedup to 69 MFlops. To get even more the ratio of floating-point operations to loads and stores must be improved (cache optimization is of hardly any use because $MVK()$, the most space-consuming item, is loaded only once). Usually this is achieved by outer loop unrolling, but that cannot be done here in a straightforward manner because of the dependency of the inner loop on M . Here the fourth property from above is of great importance. Fig. 4 shows the access pattern for $F()$. Many of the array elements are loaded more than once, but not in a way that enables register reuse.

Without further knowledge about the CPU architecture one would naively ‘transpose’ the access pattern, thereby enabling optimal register reuse (Fig. 5). Effectively, the inner and outer loops would be interchanged at the cost of introducing a third loop level. One element of $F()$ must then be loaded and stored exactly once instead of up to IQM times, so nearly half of the memory references in the loop can be saved. Access to the index field $MVK()$ is now strided, but this is not a problem due to the smallness of IQM .

The performance of this seemingly optimal solution is unsatisfactory, though. The inner loop length of at most IQM is too short for the Hitachi CPU.

A mixture of both variants is thus in order. The new outer loop (IQM blocks) is kept, but inside one block the access pattern is left as it was in the

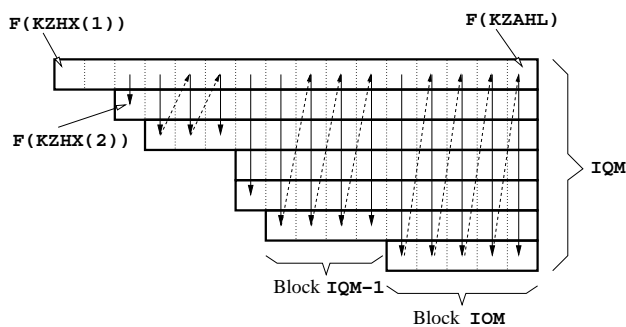


Fig. 5. Naive optimization of the first loop. Registers can be used for F() in an optimal way. A third loop level is required for the blocking pattern.

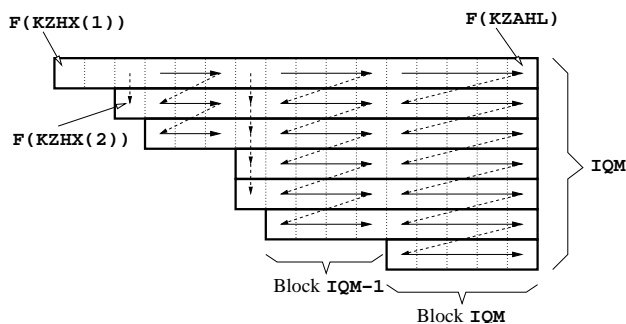


Fig. 6. Optimized version of the first loop. Unrolling of the loop over the vertical dimension is now possible.

original code (Fig. 6). Now the compiler should be able to unroll the loop over the ‘vertical’ dimension, but this is not done automatically and cannot be enforced by compiler directives. Consequently, the loop was unrolled manually by a factor of two:

```

do M=1, IQM                                     1
  ISTART=KZHX (M)                               2
  if (M.NE. IQM) then                            3
    IEND=KZHX (M+1) -1
  else                                           5
    IEND=KZAHL
  endif                                         7
  IS=1
  if (btest (M,0)) then                          9
*voption nopreload (S)
*voption noprefetch (S)                       11
    do K=ISTART, IEND
      F (K)=F (K)*S (MVK (K, IS))                13
    enddo
  enddo

```

```

        ENDDO
        IS=IS+1
    ENDF
    do MM=IS,M,2
*voption nopreload(S)
*voption noprefetch(S)
        do K=ISTART,IEND
            F(K)=F(K)*S(MVK(K,MM))*S(MVK(K,MM+1))
        ENDDO
    ENDDO
ENDDO

```

With those optimizations the overall performance of the subroutine grows to 77 MFlops. It is worth noting that the compiler now actually unrolls the middle loop by three. Further automatic optimizations like modulo variable expansion and prefetching lead to an overall unrolling factor of 48. Moderate integer register spill occurs, but limiting compiler-induced unrolling to prevent this does not improve performance any further.

3.2 Second Loop

As mentioned above, disabling pseudo-vectorization for WERTT() is the first step here. This gives a significant speedup to 87 MFlops, but there is more potential for optimization. Use of functional units in the CPU can be improved if successive iterations store their results in *different* targets. The compiler usually does loop unrolling and modulo variable expansion to achieve this, but here manual intervention is necessary.

Again, unrolling the loop by two is the method of choice. Alternating iterations write their results into the different arrays WERTT() and WERTT2():

```

41  IS=1
    IM=1
    if(bttest(KZAHL,0)) then
        WERTT(KVK(IS))=WERTT(KVK(IS)) + F(IS)
        IM=KVK(IS)
        IS=IS+1
    ENDF
    IN=IM
*voption noprefetch(WERTT,WERTT2)
*voption nopreload(WERTT,WERTT2)
    DO 50 K=IS,KZAHL,2
        WERTT(KVK(K)) = WERTT(KVK(K)) + F(K)
*voption predicate
        if(IM.lt.KVK(K)) IM=KVK(K)
        WERTT2(KVK(K+1)) = WERTT2(KVK(K+1)) + F(K+1)
*voption predicate
        if(IN.lt.KVK(K+1)) IN=KVK(K+1)

```

Table 2. Performance improvements on the Hitachi system due to the optimizations described in the text.

Version	MFlops MAT()	MFlops appl.
vanilla	26.0	26.0
1st loop S() opt.	69.4	67.8
1st loop perfect	76.7	74.6
2nd loop WERTT() opt.	86.6	83.6
2nd loop perfect	93.2	89.6

```

50    CONTINUE                                     18
      IQ=MAX(IM,IN)
*voption noprefetch(WERTT,WERTT2)                20
*voption nopreload(WERTT,WERTT2)
      do k=1,IQ                                     22
        WERTT(K)=WERTT(K)+WERTT2(K)
      enddo                                         24

```

In lines 12 and 15 the two arrays are updated. After the loop a reduction operation collects the results into `WERTT()` (starting at line 22). To do this it is required to calculate the largest index stored in `KVK()`. The FORTRAN intrinsic function `MAX0` is, however, unsuitable performance-wise. The maximum is calculated ‘on the fly’ instead (lines 5, 14, 17 and 19) by predicated assignments. This version now achieves a performance of 93 MFlops.

Of course the reduction increases the number of floating point operations in the code, but due to the smallness of `IQ` this effect is negligible.

3.3 Remarks

Tab. 2 gives an overview to the effects of the different optimizations. The performance of the whole program could be raised to 90 MFlops. The lesson to be learned is that the pseudo-vectorization features of the Hitachi CPU are not always beneficial and can sometimes even hurt performance. By pinpointing such problems and introducing some simple compiler directives the situation can be improved significantly, however.

It should be stressed that the achieved performance increase is based heavily on the peculiarities of the data structures in the code. If, for instance, only a few small values and a lot of large values (close to `KZ AHL`) were contained in `KZHX()`, the inner level of the first loop would be very short in the majority of cases which would lead to bad performance on the SR8000. Increasing the length of the array `S()` to a size comparable with the cache size would also render the prevention of `preloads` useless.

This code is actually part of a larger MPI-parallel program which typically runs on eight nodes of the SR8000-F1 at LRZ. The performance of the

Table 3. Performance improvement with the different code versions on an SGI Origin 3400 (described in section 1. ‘Perfect’ refers to the fastest version on the SR8000.

Version	MFlops appl.
vanilla	97
1st loop naive	132
1st loop naive, 2nd perfect	149
1st loop perfect, 2nd perfect	123

complete application with real-world data sets increased about twofold due to the implemented optimizations.

3.4 Comparison with SGI Origin 3000 architecture

On a CPU with less highly developed pseudo-vectorizing abilities it can be expected that the described optimizations yield smaller performance improvements. This is indeed the case, as a comparison with the SGI architecture reveals. Interestingly, the optimal code for this CPU is the naive approach with maximum register reuse. Due to the lack of `preload` functionality, the compiler does not generate pseudo-vectorized data streams for `S()` and `WERTT()` from the start. It is, however, able to unroll the middle loop over the horizontal direction in Fig. 5 to make the loop body fatter. Performance data is as indicated in Tab. 3. Hardware counter analysis yields an execution rate of two instructions per cycle with the best code version, a value that can not realistically be improved any further.

4 SMP Parallelization of a Strongly Implicit Solver

4.1 Incomplete LU-decomposition – SIP-solver

In this section a strongly implicit solver according to Stone [8] is taken as an example to demonstrate the performance impact of different implementations on the Hitachi SR8000 and other architectures. The algorithm consists of 4 parts, namely the LU-decomposition, which has to be done once, calculation of the residual, forward and backward substitution. The last three steps are performed until the residual is small enough.

All parts of the algorithm except for the calculation of the residual, contain certain data dependencies. As an example, the straightforward way to do the backward substitution looks as follows:

```
do k=2, kMaxM
  do j=2, jMaxM
    do i=2, iMaxM
```

```

RES(i, j, k) = (RES(i, j, k) - LB(i, j, k) * RES(i, j, k-1) -
*           LW(i, j, k) * RES(i-1, j, k) -
*           LS(i, j, k) * RES(i, j-1, k)) * LP(i, j, k)  6
    enddo
enddo
enddo 8

```

Three do-loops iterate over a couple of arrays containing the required data for the update of RES. Obviously, before point (i, j, k) is calculated, the points $(i-1, j, k)$, $(i, j-1, k)$ and $(i, j, k-1)$ have to be processed in advance. As a result there is no parallelization possible initially.

4.2 Hyperplane Version

A closer look at the data dependencies leads to a well-known vectorizable version [8]. Given a point (i, j, k) in the so-called “hyperplane” $L = i + j + k = \text{const.}$, only accesses to points in planes $L \pm 1$ are necessary. Having this in mind, one can proceed through all points of a hyperplane, creating long loops with no dependencies. This makes this kind of implementation especially suitable for vector machines. However, by using hyperplanes each point implies an indirect and non-contiguous memory access (see Fig. 7).

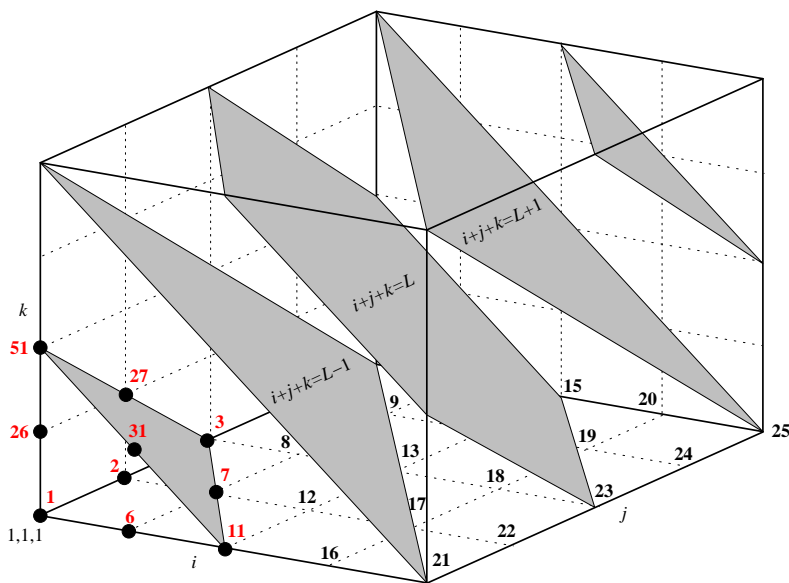


Fig. 7. Hyperplanes with $L = i + j + k = \text{const.}$ in a cube.

In addition, this version allows for a parallelization of the calculation of all points *within* one hyperplane. The hyperplanes have to be updated one after another, however. The following code fraction shows the forward substitution:

```

do l=1,hyperplanes                                1
    n=ICL(1)                                       3
    do m=n+1,n+LM(1)                              5
        ijk=IJKV(m)
        RES(ijk)=(RES(ijk)-LB(ijk)*RES(ijk-ijMax)-LW(ijk)* 7
        *          RES(ijk-1)-LS(ijk)*RES(ijk-iMax))*LP(ijk)
    enddo                                          9
enddo

```

The inner loop is parallelized using appropriate compiler/OpenMP directives.

4.3 Hyperline Version

One could as well define the hyperplanes by $L = i + k = \text{const.}$, making them parallel to the j direction. The term “hyperline” was coined to illustrate this strategy. Again, for constant L only points in $L \pm 1$ are accessed, enabling simple shared memory parallelization within the hyperline. This is shown in Fig. 8. The forward substitution looks similar to the hyperplane version:

```

do l=1,hyperlines                                2
    do diag = startStopDiag(1,1),startStopDiag(2,1)  4
        k=coordDiag(1,diag)
        i=coordDiag(2,diag)                        6
        do j=2,jMaxM                              8
            ijk=(k-1)*ijMax+(i-1)*jMax+j
            RES(ijk)=(RES(ijk)-LB(ijk)*RES(ijk-ijMax)- 10
            *          LW(ijk)*RES(ijk-1)-LS(ijk)*
            *          RES(ijk-jMax))*LP(ijk)        12
        enddo
    enddo                                          14
enddo

```

The advantage is that now all points in j -direction are accessed continuously, leading to increased data locality. The full content of a cache line that is loaded from memory can be used at least once before the next one is accessed. This is usually not the case for the hyperplane version, where every load from memory is punished with latency.

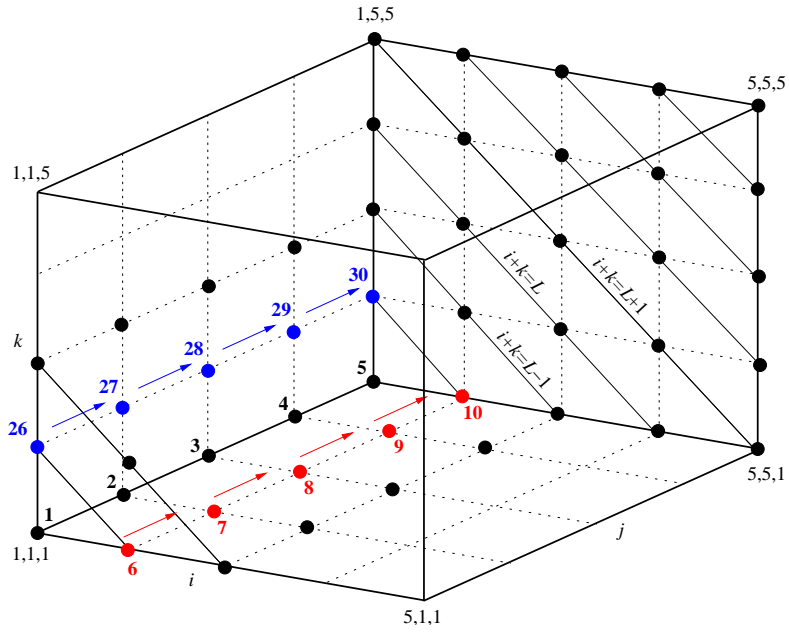


Fig. 8. Hyperlines with $L = i + k = \text{const.}$ in a cube.

4.4 Pipeline Parallel Processing

Going back to the straightforward version described in section 4.1, Hitachi’s Fortran90 compiler is capable of so-called *pipeline parallel processing* (Fig. 9). Parallelization is done for the loop along the i direction (the middle loop), but calculation of any chunk is delayed by a barrier until the chunk left of it is processed. Consequently, blocks with equal color in Fig. 9 are calculated concurrently. This leads to load imbalance in the “windup” and “winddown” phases of this pipeline, but this effect is negligible for a large enough lattice.

The difference to the hyperline version is that the parallelization is done for the (3D) middle loop instead of a hyperline. As indicated by the arrows inside the rightmost ‘C’ block in Fig. 9, outer loop unrolling is possible and reduces the load-to-flop ratio.

4.5 Results and Comparison

All experiments were performed with a benchmark code whose basic core was obtained from [9]. It decomposes a given matrix once and performs a fixed number of iterations depending on problem size (10000–500 for size 31^3 – 91^3). Considering runtime, the LU-decomposition is therefore insignificant. Memory consumption for problem size 91^3 is about 100 MB.

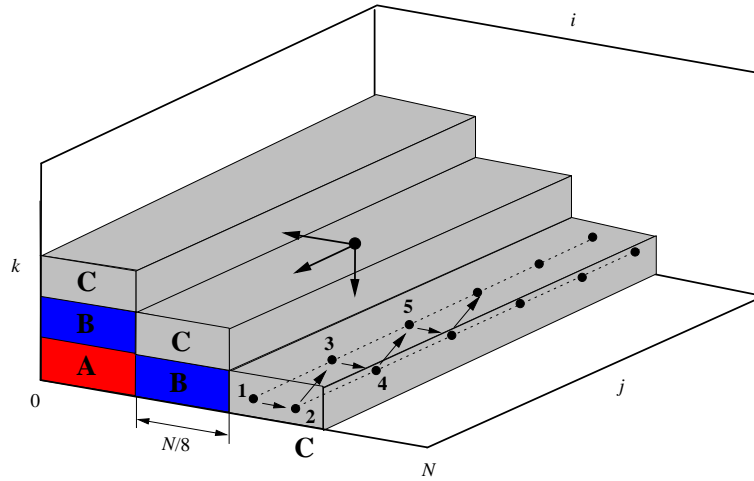


Fig. 9. Schematic view of pipeline parallel processing. Blocks with equal colors (or letters) are calculated simultaneously. One CPU always processes the points inside a certain range of the middle (i) loop.

Fig. 10 shows the performance on the Hitachi SR8000 for one CPU and one node for different problem sizes. Obviously the pipeline parallel processing

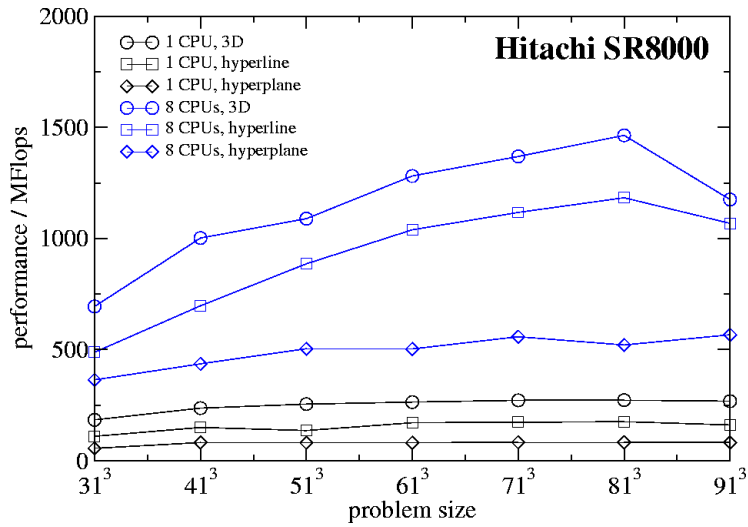


Fig. 10. Hitachi SR8000: performance for SIP-solver variants.

version (3D) performs best and the vectorized hyperplane version is worst.

Fig. 11 contains performance data for all the platforms under consideration, and the fastest code variant was chosen for each. The hyperplane version

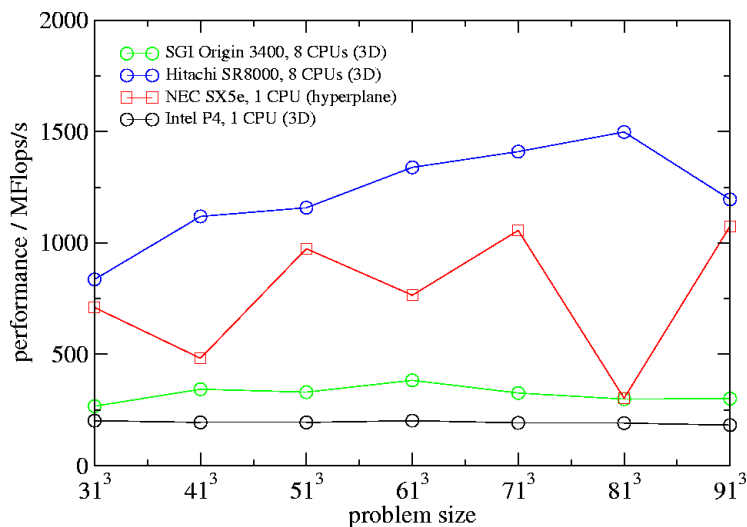


Fig. 11. SIP-solver benchmark: performance on different architectures. The fastest code was chosen on each machine.

shows outstanding performance on the NEC SX5 for a single CPU run. This version is highly suited for vector architectures as it provides long vector lengths within hyperplanes (there are periodic degradations of the performance which might emerge from disadvantageous access patterns and have not been investigated further). Nevertheless, the 3D version on one Hitachi node outperforms all other versions by far.

4.6 Summary

A straightforward port of the hyperplane version to the Hitachi, exploiting the PVP-feature, yields lower performance than the pipeline parallel processing version. In spite of the `preload` mechanism, indirect memory access and varying loop lengths make for slower code. Pipelined parallelism, on the other hand, provides the possibility for outer loop unrolling and cache reuse and is thus the best choice on this machine.

5 Conclusions and Acknowledgments

The Hitachi SR8000 system shows performance characteristics which are similar to comparable vector architectures. A vector-like programming style is

thus usually a good guess at first, but when cache locality and classic RISC optimization strategies are applicable, it is by all means advisable to use those features. Sometimes, however, one must rely on more uncommon techniques to obtain really outstanding performance.

We would like to thank the HPC team at the LRZ for ongoing support and fruitful discussions. This work was supported by the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR).

References

1. Available at <http://www.top500.org/>
2. K. Shimada, T. Kawashimo, M. Hanawa, R. Yamagata and E. Kamada: *A Superscalar RISC Processor with 160 FPRs for Large Scale Scientific Processing*. Proc. of International Conference on Computer Design (1999), pp. 279-280
3. W. Schönauer: *Architecture and Use of Shared and Distributed Memory Parallel Computers*, eds.: W. Schönauer, ISBN 3-00-005484-7.
4. G. Wellein, G. Hager, A. Basermann, and H. Fehske, in *Proceedings of VEC- PAR2002*, Porto (2002).
5. R. Barrett et al.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia (1993).
6. M. Kinateder, G. Wellein, A. Basermann, and H. Fehske, in *High Performance Computing in Science and Engineering '00*, edited by E. Krause and W. Jäger, Springer-Verlag, Berlin Heidelberg (2001), pp. 188-204.
7. B. Pfitzinger, H. M. Hofmann and G. M. Hale: *Elastic p - ^3He and n - ^3H scattering with two- and three-body forces*. Phys. Rev. C **64** (2001) 044003
8. J. H. Ferziger, M. Perić: *Computational Methods for Fluid Dynamics*. Springer Verlag, 1999
9. Basic code examples for the algorithms in [8] can be obtained from <ftp://ftp.springer.de/pub/technik/peric/>