Fast sparse matrix-vector multiplication for TeraFlop/s computers

G. Wellein[1], G. Hager[1], A. Basermann[2] and H. Fehske[3]

[1] Regionales Rechenzentrum Erlangen, D-91058 Erlangen, Germany
[2] C&C Research Laboratories, NEC Europe Ltd, D-53757 Sankt Augustin, Germany
[3] Institut für Physik, Universität Greifswald, D-17487 Greifswald, Germany

Abstract. Eigenvalue problems involving very large sparse matrices are common to various fields in science. In general, the numerical core of iterative eigenvalue algorithms is a matrix-vector multiplication (MVM) involving the large sparse matrix. We present three different programming approaches for parallel MVM on present day supercomputers. In addition to a pure message-passing approach, two hybrid parallel implementations are introduced based on simultaneous use of message-passing and shared-memory programming models. For a modern SMP cluster (HITACHI SR8000) performance and scalability of the hybrid implementations are discussed and compared with the pure message-passing approach on massively-parallel systems (CRAY T3E), vector computers (NEC SX5e) and distributed shared-memory systems (SGI Origin3800).

1 Introduction

Clusters of multiprocessor shared-memory (SMP) nodes are presently the common way to exceed the TFlop/s barrier. Although these systems outperform all other architectures with respect to peak performance there is still the question how to write portable codes which run and scale efficiently. The simplest approach is to use massively-parallel (MPP) codes totally ignoring the shared-memory within SMP nodes. A more promising approach seems to be the "hybrid programming model" which uses both message-passing model (between SMP nodes) and shared-memory model (within SMP nodes) simultaneously and maps most closely to the memory structure of the underlying target architecture. It is evident that the benefit of either programming model depends on the algorithm and the programming style used. Some studies for application programs have already been done in this context [1–6] leading to different results.
Numerical cores of many applications are iterative sparse matrix algorithms such as sparse eigenvalue solvers. In practice, such operations will have a performance bound by the memory access. Since the LINPACK benchmark [7] usually does not cover this quantity, an iterative solver benchmark (*sparsebench*) has been developed recently by Dongarra et al. [8]. The *sparsebench* benchmark solves sparse linear systems using different methods and storage schemes but is available in a non-parallel version for small problem sizes only.
The aim of this work is to discuss three different parallel implementations of a sparse matrix-vector multiplication (MVM) for SMP clusters:

MPP-mode : Each MPI process is assigned to exactly one processor of a SMP node. This implementation makes no use of the hardware structure.

Vector-mode : One SMP node is identified as a single process with high peak-performance and memory bandwidth similar to a single vector processor. Shared-memory parallelisation of long vector loops is done. This implementation might be beneficial for vector codes on SMP nodes providing fast collective thread operations.

Task-mode : Using OpenMP we perform an explicit programming of each processor of a SMP node to overlap communication and computation. An asynchronous data transfer can be simulated by defining one processor to be responsible for communication. This strategy improves performance on SMP nodes without hardware support for asynchronous data transfer and for codes where a sufficient overlap of communication and computation exists.

The different implementations have been applied to extreme sparse matrices (originating from large scale eigenvalue problems in theoretical physics) with dimensions ranging from 10^3 to 10^8 and have been run on a maximum of 1024 processors. For completeness a detailed performance comparison with classical supercomputers is given.

2 Implementation of matrix-vector multiplication

Generally, the performance of iterative sparse eigenvalue solvers is determined by a matrix-vector multiplication step involving the large sparse matrix. There are two widely used methods to exploit the sparsity of the matrix to minimize the memory requirements in the MVM step which show different performance characteristics. The *Compressed Row Storage* (CRS) format [9] is generally used for RISC systems but has serious drawbacks on vector processors due to short inner loop lengths. The *Jagged Diagonal Storage* (JDS) scheme [9] provides long inner vector loops at the cost of minor performance drawbacks on RISC systems for matrices used in our work [10]. To achieve high performance on a wide range of supercomputer architectures, a parallel JDS scheme was implemented.

2.1 JDS matrix-vector multiplication

The JDS format is a very general storage scheme for sparse matrices making no assumptions about the sparsity pattern. In a compression step, the zero elements are removed, nonzero elements are shifted to the left and the rows are made equal in length by padding them with trailing zeros. To prevent storing theses trailing zeros, the rows are rearranged according to the number of nonzero elements per row. The columns of the compressed and permuted matrix are called *Jagged Diagonals*. Obviously, the number of these *Jagged Diagonals* is equal to the maximum number `max_nz` of nonzero elements per row. The MVM is performed along the *Jagged Diagonals*, providing an inner loop length essentially equal to the matrix dimension (D_{Mat}). For a detailed description of our JDS implementation we refer to Ref. [10].
Using permuted vectors, the numerical core of the MVM is given as follows:

```
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 nonzero elements of A are stored in the linear array `value(:)`, stringing the *Jagged Diagonals* together one by one. An array of the same length (`col_ind(:)`) contains the corresponding column indices. The array indices belonging to the first element of each *Jagged Diagonal* are stored in the pointer array `jd_ptr(:)`. The performance of the innermost loop is clearly determined by the quality of the memory access: One `store` and `four` load operations are required to perform two floating point operations (Flop). Taking into account the gather operation on the vector `x`, we do not expect MVM performance to exceed 20–25 % of the peak performance even for present-day vector processors delivering a memory bandwidth of one word per Flop. The performance on RISC systems is even worse [8].

2.2 Parallel implementation

For parallel computers with distributed memory, we use a row-wise distribution of matrix and vector elements among the processes (**MPP-mode**) involved in MVM (cf. Fig. 1 (left panel)). Matrix elements requiring access to vector data

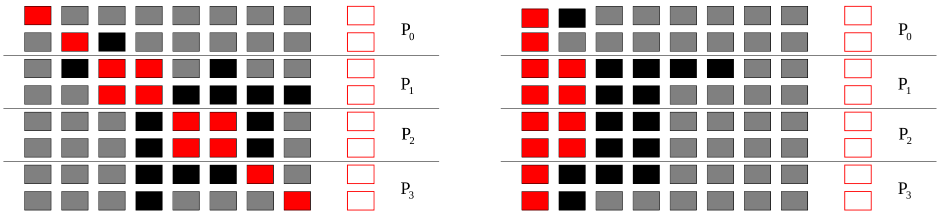


Fig. 1. Distribution of matrix and vector elements on 4 processes ($P_0 - P_3$). Nonzero matrix elements causing communication are marked black and local nonzero matrix elements are colored red. The original distribution is given in the left panel. The right panel depicts the distribution after the modified JDS compression step.

stored on remote processes (nonlocal vector elements) cause communication. Since the matrix does not change during eigenvalue computation, a definite communication scheme is constructed once, allowing an efficient way of exchanging data as well as an overlapping of communication and computation in the MVM step.
Each process transforms its local rows into the JDS format as described above (see also Ref. [10]). In order to maintain a MVM core similar to that shown above, local and nonlocal vector elements of `x` are stored in the same array. It is apparent that a *Jagged Diagonal* can only be released for computation if the

corresponding nonlocal vector elements were successfully received.
The parallel MVM implementation is based on MPI and uses non-blocking communication routines. After all communication calls have been initialized (pairs of `mpi_isend` and matching `mpi_irecv`), we check repeatedly for the completion of any receive call (`mpi_testany`) and release *Jagged Diagonals* for computation if possible. In principle this implementation allows overlapping of communication and computation if asynchronous data transfer is supported by the hardware and at least one local *Jagged Diagonal* exists, involving local matrix elements only. At this point the parallel JDS scheme as described above has one serious drawback: The (process-)local compression step might cause a strong mixture of local and nonlocal matrix elements when shifting the nonzero elements to the left, e.g. after the JDS transformation in Fig. 1 (left panel) process P_3 requires at least one receive-operation to be completed before starting any computation. For that reason we have slightly modified the parallel compression step as follows:

1. Shift local matrix elements to the left only within the local sub-block on each process.
2. Fill each local sub-block with the nonlocal matrix elements from the right.

If the matrix-diagonal is fully occupied the algorithm generates at least one local *Jagged Diagonal* as can be seen in Fig.1 (right panel).

2.3 Hybrid parallel implementation

The MVM implementation presented above allows the efficient use of vector processors since the long inner loop can easily be vectorised by the compiler. The same holds for vector parallel computers where the loop length decreases with increasing process number but in general remains long with respect to vector start-up times.
Of course, the inner loop can also be automatically parallelised and executed in parallel on the processors of a SMP node. Thus we can easily use our MPI based implementation on SMP clusters, assigning one MPI process to each node and running a shared memory parallelisation within each node. When compared to the **MPP-mode**, this hybrid approach (**vector-mode**) provides both a lower memory consumption and less communication overhead due to a reduced number of MPI processes. However, there may be one serious drawback: If we assume that MPI routines are called by the master thread only and that there is no asynchronous transfer of the data, all slave threads are blocked during data transfer even if *Jagged Diagonals* could be released for computation.
A hybrid programming approach based on an explicit programming of each thread (**task-mode**) can resolve that problem by using one thread for *asynchronous data transfer*. We choose the master thread to do communication while slave threads perform computations. The list of released diagonals is governed by the master thread and is protected with a lock mechanism. Each slave thread computes a fixed number of local rows to avoid the synchronisation of the slave threads at the end of the inner loop. Due to different length of the *Jagged Diagonals* an equal distribution of rows may result in a load imbalance for slave threads. Instead we use an equal distribution with respect to the number of

floating point operations to be performed on each slave thread. For portability reasons, we have chosen an OpenMP implementation for the **task-mode**:

```
!$OMP PARALLEL
thread_id = OMP_GET_THREAD_NUM()
if( thread_id .eq. 0) then
    1.) release local diagonals
    2.) start receive and send operations
    3.) wait for the completion of any receive operation
    4.) release diagonals for computation if possible
    5.) if there is still an outstanding receive operation goto 3.)
    6.) wait for the completion of all send operations
else
    1.) wait until at least one diagonal is released for computation
    2.) compute MVM for all released diagonals
    3.) if not all diagonals have been released goto 1.)
endif
!$OMP END PARALLEL
```

At this point the question arises, which scenarios favor the use of **task-mode** or **vector-mode**. The basic performance relationship between the hybrid modes can be determined qualitatively using a simple model: Assuming a perfect load balance concerning the MPI processes, a full speed up with respect to the number of threads (N_{THR}) within a SMP node as well as no MPI communication hiding in the **vector-mode**, the total runtime per node for the **vector-mode** can be written as

$$T_{\mathbf{VECTOR}} = T_{\mathrm{COMP}}/N_{\mathrm{THR}} + T_{\mathrm{COMM}} \tag{1}$$

where T_{COMM} denotes the time spent for MPI communication and T_{COMP} is the compute time using one thread per node. Furthermore, in **task-mode** we assume that MPI communication can fully be overlapped by computation giving a total runtime which is determined solely by T_{COMP} with a reduced number of threads:

$$T_{\mathbf{TASK}} = T_{\mathrm{COMP}}/(N_{\mathrm{THR}} - 1) \geq T_{\mathrm{COMM}} \tag{2}$$

The performance ratio $\epsilon_{\mathrm{T,V}}$ of **task-mode** to **vector-mode** is given by the inverse ratio of the corresponding runtimes and can be written as follows:

$$\epsilon_{\mathrm{T,V}} = \frac{T_{\mathbf{VECTOR}}}{T_{\mathbf{TASK}}} = (1 - \frac{1}{\mathrm{N_{THR}}}) + (N_{\mathrm{THR}} - 1)\frac{T_{\mathrm{COMM}}}{T_{\mathrm{COMP}}} \tag{3}$$

It is evident that the minimum value of $\epsilon_{\mathrm{T,V}}$ is adopted in the limit of vanishing communication ($T_{\mathrm{COMP}} \gg T_{\mathrm{COMM}}$) where the number of compute threads determines the runtimes. The upper limit is given by Eq. (2) yielding to:

$$1 - 1/N_{\mathrm{THR}} \leq \epsilon_{\mathrm{T,V}} \leq 2 - 1/N_{\mathrm{THR}} \tag{4}$$

Obviously there is a crossover between problems with low communication intensity ($T_{\mathrm{COMM}}/T_{\mathrm{COMP}} \ll 1$) where the **vector-mode** is the proper choice and more complex problems which benefit from the use of the **task-mode**. In the limit $T_{\mathrm{COMM}} \gg T_{\mathrm{COMP}}$, however, Eq. (2) does not hold and the hybrid modes show approximately the same performance as the total runtime is determined mainly by communication. To sum it up, the **task-mode** benefits from larger thread counts and can provide a performance gain of up to 100% compared to the **vector-mode**. This effect is expected to be most pronounced if communication time and computation time are of similar size.

3 Benchmark Issues

3.1 Benchmark platforms

The benchmarks have been performed on the platforms given in Table 1. While

Table 1. Benchmark platforms and single processor specifications. LINPACK (cf. Ref. [7]) and peak performance numbers (Peak) are given in GFlop/s, whereas the memory bandwidth (memb) per processor is given in GBytes/s. The rightmost columns contain the sizes of the L1 and L2 cache for the RISC processor based systems.

Platform	Installation site	#CPU	LINPACK	Single CPU specifications			
				Peak	memb	L1 [kB]	L2 [MB]
NEC SX5e	HLR Stuttgart	32	121	4.0	32.0	—	—
CRAY T3E-1200	NIC Jülich	540	447	1.2	0.6	8	0.096
SGI Origin3800	TU Dresden	128	n.a.	0.8	0.8	32	8
HITACHI SR8000	LRZ Munich	168 × 8	1645	1.5	4.0	128	—

architecture and memory hierarchy of the RISC based systems (SGI/CRAY) are widely known, some important features of the NEC vector computer and the HITACHI SR8000 should be emphasized.

The NEC SX5e vector processor runs at a frequency of 250 MHz using eight-track vector pipelines. There is one load and one load/store pipeline which can either work in parallel as two load pipelines or as a single store pipeline.

The HITACHI SR8000 processor is basically an IBM PowerPC processor running at 375 MHz with important modifications in hardware and instruction set. 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, the `preload` instruction can load single data items directly from the memory to the large 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 the main memory to the processor avoiding the penalties of memory latency. Combining `prefetch` and preload instructions, even vector-gather operations such as the indirect access to vector x in the MVM (cf. Sect. 2.1) can be executed in a vector processor like manner (cf. Fig. 2). The prefetch transfers a full cache-line (16 or 32 data

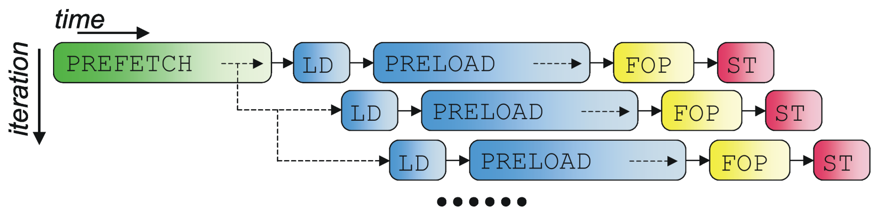


Fig. 2. Diagrammatic view of Pseudo-Vector-Processing for indirect memory access

items depending 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 even for non-contiguous memory access. Eight processors form a shared memory node with a memory bandwidth of 32 GByte/s which is basically the same as for a single NEC SX5e processor. In contrast to the NEC SX5e this bandwidth holds for load and store operations. Hardware support for fast collective thread operations is provided. Therefore the system can efficiently be used either as a vector-like computer running one process comprising eight threads on one node or as an MPP assigning one process to each processor.

3.2 Benchmark matrices

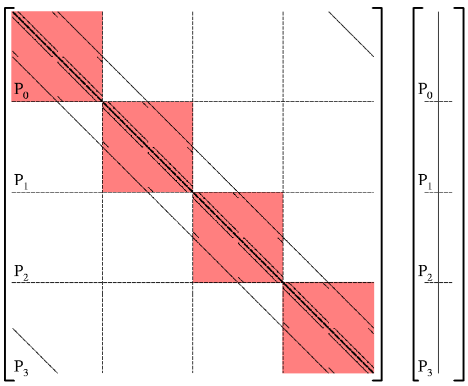
Two different types of matrices have been chosen for the performance evaluation. As a first example, we consider a modeling of disordered solid by a finite-dimensional Anderson Hamiltonian matrix [12]

$$H = \sum_i \epsilon_i |i\rangle\langle i| - t \sum_{\langle i,j\rangle} |i\rangle\langle j| \tag{5}$$

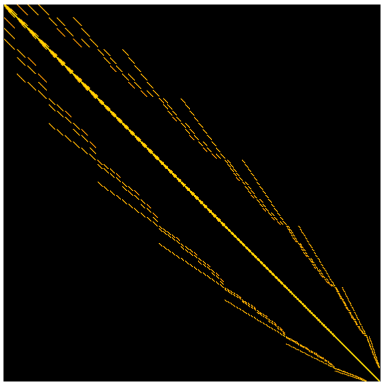
defined on a D-dimensional hypercubic lattice, where $|i\rangle$ denote tight-binding states situated at lattice point i and $\langle i,j\rangle$ means nearest neighbours only. The on-site energies $\{\epsilon\}$ are assumed to be independent, identically distributed random variables. During the last five decades, the Anderson Hamiltonian (AH) has been intensively studied as a generic model for the localization of electronic states due to disorder effects. Working in the one-particle sector, the matrix structure of the AH is identical to the seven point discretisation of the differential operator on a three dimensional Cartesian grid (periodic boundary conditions) and will be denoted as AH matrix in the following. If the conversion of cartesian coordinates ($\{i,j,k\}$) to a linear index l is defined as follows

$$\begin{aligned} \{i,j,k\} \longrightarrow l = i + (j-1)\cdot n_i + (k-1)\cdot n_i \cdot n_j \\ (i = 1,\ldots,n_i \;;\; j = 1,\ldots,n_j \;;\; k = 1,\ldots,n_k) \end{aligned} \tag{6}$$

and parallelisation using $n_{\rm proc}$ MPI processes is done along the k-direction ($n_k^{\rm loc} = n_k/n_{\rm proc}$), a matrix structure and data distribution is generated as depicted in Fig.3(a). It is well known that diagonal storage techniques are ideally



(a)



(b)

Fig. 3. (a) Structure and data distribution for `AH matrix` with $D_{\rm Mat} = 8^3$. (b) Sparsity pattern of `HHM matrix` ($D_{\rm Mat} = 0.5 \times 10^6$) with 8.6×10^6 nonzero matrix entries.

suited for this class of matrices. Nonetheless these matrices provide a valuable case study for our JDS MVM implementation because

- problem sizes and number of non-zero matrix entries ($7 \cdot n_i \cdot n_j \cdot n_k$) can linearly be scaled,
- the communication scheme is independent of problem sizes (nearest neighbour communication only),
- the number of floating point operations ($14 \cdot n_i \cdot n_j \cdot n_k^{\rm loc}$) and amount of data transfer ($n_i \cdot n_j$ data items per direction) per process can easily be controlled.

As a second example, we consider the Holstein Hubbard Hamiltonian

$$H = -t \sum_{i,\sigma} (c_{i\sigma}^\dagger c_{i+1\sigma} + \text{H.c.}) + U \sum_i n_{i\uparrow} n_{i\downarrow} + g\omega_0 \sum_{i,\sigma} (b_i^\dagger + b_i) n_{i\sigma} + \omega_0 \sum_i b_i^\dagger b_i, \quad (7)$$

which can be taken as a fundamental model for the interaction of electron and lattice degrees of freedom in solids. Here $c_{i\sigma}^\dagger$ creates a spin-σ electron at Wannier site i ($n_{i,\sigma} = c_{i\sigma}^\dagger c_{i\sigma}$), $b_i^\dagger$ creates a local phonon of frequency ω_0, t denotes the hopping integral, U is the on-site Hubbard repulsion, g is a measure of the electron-phonon coupling strength. Because the Holstein Hubbard model (HHM) describes two types of particles (fermions and bosons) and their interactions, a much more complicated matrix structure results (see Fig. 3(b)). Although the number of non-zero entries per row varies, we choose a simple row-wise distribution for parallelisation at the cost of some load imbalance.

4 Performance analysis

4.1 Sequential performance

First the sequential performance as a function of the problem size D_{Mat} is discussed (see Fig. 4) using a `HHM matrix` which basically reflects the qualitative effects. For the SGI Origin we recover the well known cache effect: At

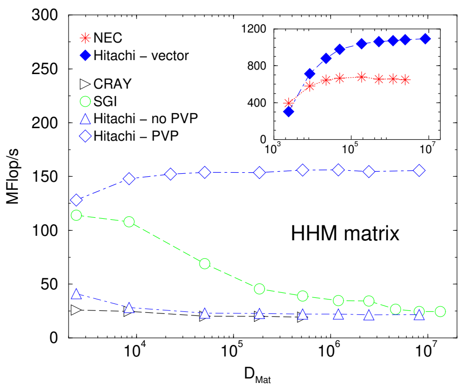


Fig. 4. Single processor performance as a function of the matrix dimension D_{Mat} (`HHM matrix`). The inset depicts a comparison of one NEC SX5e processor and one HITACHI SR8000 node running in **vector-mode**.

small problem sizes cached data can be reused giving a performance of about 110 MFlop/s. Consistent with the large L2 cache size the performance drops around $D_{\mathrm{Mat}} \sim 10^4 - 10^5$ and an asymptotic performance of about 35 MFlop/s is sustained for local memory access. If the matrix size is further increased ($D_{\mathrm{Mat}} > 2.5 \times 10^6$) **nonlocal** memory on remote compute nodes has to be allocated. Although the network could in principle sustain the same memory bandwidth for local and remote memory access, the performance decreases to 25 MFlop/s due to higher latencies for remote memory accesses. Since the problem sizes in Fig. 4 start at $D_{\mathrm{Mat}} \sim 10^3$ there is only a minor cache-effect for CRAY T3E and HITACHI SR8000 (small cache sizes). For the HITACHI SR8000 this picture changes completely if the PVP feature (cf. discussion in Sect. 3.1) is enabled. As a consequence of the vector-like processor characteristic, performance increases with increasing problem size and saturates at long loop lengths. With an asymptotic performance of about 150 MFlop/s, the HITACHI SR8000 processor is able to maintain a reasonable portion of its high memory-bandwidth and outperforms present-day RISC processors by a factor of roughly 5. Therefore PVP has always been enabled in what follows.
It is straightforward to use one HITACHI SR8000 node in **vector-mode** by enabling the automatic parallelisation compiler feature which generates a shared-memory parallelisation of the inner loop in the MVM (cf. Sect. 2.1). In the inset of Fig. 4 the MVM performance of one HITACHI SR8000 SMP node (eight processors) is shown together with the measurements for one NEC SX5e vector processor. Again the HITACHI SR8000 realizes a vector-like characteristic and saturates at a performance of about 1100 MFlop/s or 138 MFlop/s per processor. A decrease of less than 10 % compared to the single processor performance

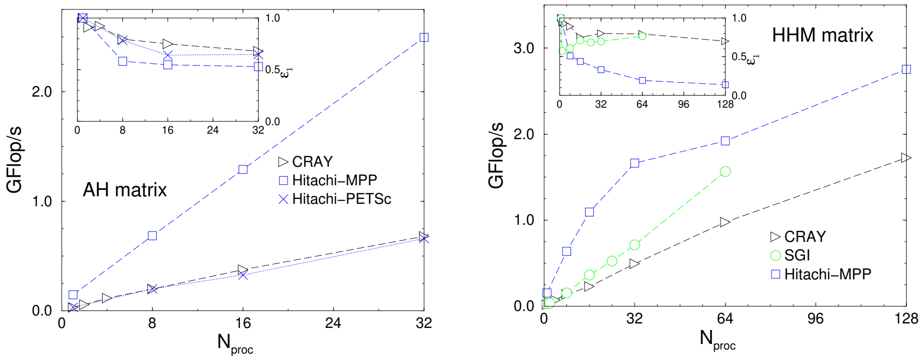
indicates only minor memory contention problems in the vector-mode. Surprisingly one SMP node exceeds the performance of the NEC SX5e at intermediate and large problem sizes although the maximum memory bandwidth is the same for both systems. Since both systems achieve approximately the same performance for vector triads (cf. Ref. [11]), this effect can not be attributed to the higher `store` bandwidth of the HITACHI SR8000 node, which is twice the `store` bandwidth of the NEC SX5e processor (cf. discussion in Sect. 3.1). Obviously the indirect load can be performed more efficient by the PVP feature than by a vector-gather instruction. However, at small problem sizes the vector processor outperforms the SMP node because of short vector pipeline start-up times.

4.2 MPP programming model

At the next step we determine the scalability of code and platforms within the **MPP-mode**, where one MPI process is assigned to each processor. For the HITACHI SR8000 we use eight MPI processes per node. In order to minimize inter-node communication a continuous assignment of processes to nodes is chosen.
As a starting point, basic questions are addressed in Fig.5(a) using `AH matrix`. Even though a rather small matrix size is considered, we find reasonable scalability measured by the parallel efficiency

$$\epsilon_1(N_{\text{proc}}) = T(1)/(N_{\text{proc}} \cdot T(N_{\text{proc}})) \tag{8}$$

($T(N_{\text{proc}})$ is the time per MVM step on N_{proc} processes) even on 32 processors on HITACHI SR8000 ($\epsilon_1(32) = 53$ %) and CRAY T3E ($\epsilon_1(32) = 68$ %). The advantage of JDS format for vector and vector like processors is illustrated



(a) (b)

Fig. 5. (a) Performance of JDS implementation and PETSc for `AH matrix` with $D_{\text{Mat}} = 64^3 = 0.26 \times 10^6$. (b) Scalability of the **MPP-mode** for different systems ($D_{\text{Mat}} = 1.2 \times 10^6$). Both insets show the parallel efficiencies.

by comparison to the sparse MVM routine provided by the PETSc library [13]

which uses CRS format. As the CRS implementation provides only short loop length for the innermost loop (number of nonzero entries per row is 7 for `AH matrix`) the PVP feature of the HITACHI SR8000 has no relevance, leading to a single processor performance of 32 MFlop/s which is about the same as for the JDS format with the PVP feature being disabled (see Fig. 4). Although the PETSc routine shows scalability numbers similar ($\epsilon_1(32) = 63$ %) to the CRAY T3E the performance gap can only be slightly reduced with increasing processor count. Basically the same scalability behaviour can be found with the `HHM matrix` at larger problem size (cf. Fig. 5(b)): We achieve best scalability for the CRAY T3E system where a parallel efficiency ϵ_1 of more than 80 % can be maintained up to 64 processors and even with 256 processors 72 % is still exceeded. Due to the higher single processor performance (cf. Fig. 4) the SGI Origin outperforms the CRAY T3E at low processor counts. Interestingly this performance gap widens at intermediate processor counts accompanied by an increase in the parallel efficiency for the Origin system [14]. This fact can be attributed to the large amount of aggregate L2 cache size with increasing number of processors involved in the MVM, e.g. the total L2 cache of 16 processors can hold all nonzero elements of the matrix used in Fig. 5(b).
The worst scaling is again provided by the HITACHI SR8000. Even when intra-node communication is used only (up to $N_{\mathrm{proc}} = 8$), a sharp drop in the efficiency occurs. Going beyond eight processors, both intra- and inter-node communication is used and a steady decrease in efficiency was measured. Mainly two reasons account for the poor scalability of the HITACHI SR8000: First the ratio between inter-node communication bandwidth (1 GByte/s) and local memory bandwidth (32 GByte/s) is not as balanced as for the CRAY T3E and SGI Origin systems. Second, the MPI performance suffers a drop if a large number of outstanding messages has to be governed [15]. Nevertheless, the HITACHI SR8000 provides the best absolute performance values due to the excellent single processor performance. At fixed problem size, however, the SGI Origin benefits from the scalability both of interconnect and code as well as the aggregate L2 cache and narrows the gap with increasing processor count.

4.3 Hybrid programming model

It is obvious to improve the HITACHI SR8000 performance by reducing the scalability problems. The two hybrid parallel programming approaches (**vector-mode** and **task-mode**) introduced in Sect. 2.3 seem to be feasible in this context. Using the parameters of Fig. 5(b) a comparison between **MPP-mode** and hybrid modes as well as NEC SX5e is depicted in the inset of Fig. 6. We find that the hybrid modes ($\epsilon_1(12) \approx 26$ %) scale significantly better than the **MPP-mode** ($\epsilon_1(12) = 16.9$ %) resulting in a performance improvement of up to 50 % at 12 nodes. The same holds even if the problem size is increased by a factor of 10 (cf. Fig. 6). Obviously the hybrid modes benefit both from the reduction of the number of outstanding MPI messages and from an improvement in load balancing by an increase in the number of local rows per MPI process. Best

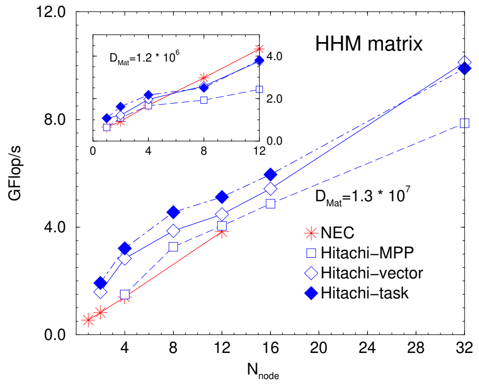


Fig. 6. Hybrid programming performance at fixed matrix dimensions D_{Mat} as a function of SMP nodes (N_{node}). For comparison the HITACHI SR8000 MPP (one node equals eight MPI processes) and NEC (one node equals one vector processor) measurements are depicted.

scalability, however, is found for the NEC SX5e system where the MPI communication is done within a shared-memory. Achieving parallel efficiencies of more than 50 % up to 12 processors, the NEC SX5e can close or even overcome the single processor performance gap (cf. Fig. 4).
A comparison of the two hybrid modes qualitatively confirms the discussion in Sect. 2.3: The performance gain of the **task-mode** is most pronounced at low and intermediate node counts and vanishes at a large number of nodes. Increasing the problem size widens the performance gap and shifts the crossover to higher node counts.
The qualitative picture as described in the context of `HHM matrices` also holds for `AH matrices` (cf. Fig. 7), whereas the absolute performance figures for a fixed matrix size significantly increase at intermediate and large node counts due to the reduced complexity in the `AH matrix` structure. Moreover, the `AH matrices`

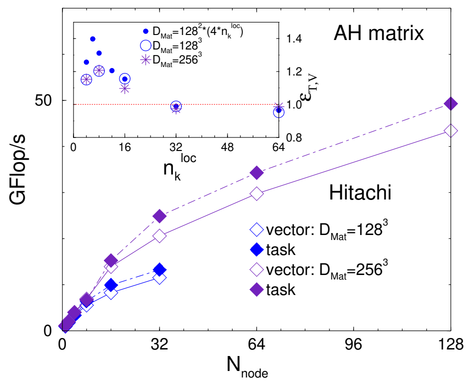


Fig. 7. Hybrid programming performance on HITACHI SR8000 (`AH matrix`) as a function of SMP nodes (N_{node}). The corresponding performance ratio $\epsilon_{\mathrm{T,V}}$ is depicted in the inset together with additional data for a fixed number of nodes (N_{node}=4) but varying n_k^{loc} as well as the crossover line $\epsilon_{\mathrm{T,V}} = 1$ (red line)

allow a detailed analysis of the performance ratio $\epsilon_{\mathrm{T,V}}$ of **task-mode** to **vector-mode** since the dependency between between $T_{\mathrm{COMM}}/T_{\mathrm{COMP}}$ and the matrix parameters is given by the ratio of floating point operations per process to amount of data transfer per process (cf. Sect. 3.2.) and can be written as

$$\frac{T_{\mathrm{COMM}}}{T_{\mathrm{COMP}}} \propto \frac{n_i \cdot n_j}{n_i \cdot n_j \cdot n_k^{\mathrm{loc}}} = \frac{1}{n_k^{\mathrm{loc}}}. \tag{9}$$

Following Eq.(3) $\epsilon_{\mathrm{T,V}}$ should mainly be dominated by the variation of the local workload per process (n_k^{loc}). This result is in good agreement with data shown in the inset of Fig. 7 where $\epsilon_{\mathrm{T,V}}$ is nearly independent of matrix size or number of nodes being used. Thus we are able to locate a fixed crossover point which determines when to use **task-mode** ($n_k^{\mathrm{loc}} \leq 32$) or **vector-mode** ($n_k^{\mathrm{loc}} \geq 32$). Furthermore we find that speed-up values of typically 15–30 % are achieved with the **task-mode** which is in the range given by Eq. (4). Finally, we clearly demonstrate in Fig. 8 that only the hybrid modes provide reasonable performance for large scale matrices used in our work. In this context a scalability

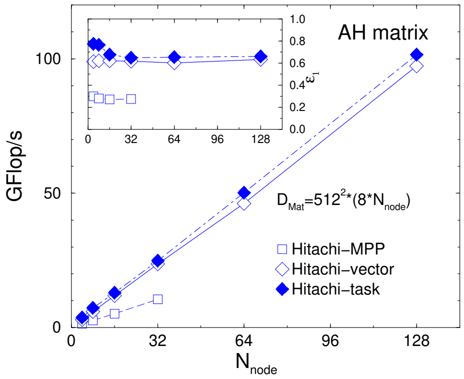


Fig. 8. Performance and parallel efficiency on the HITACHI SR8000 (`AH matrix`) as a function of SMP nodes (N_{node}) using a fixed problem size per processor ($n_i = n_j = 512$; $n_k^{\mathrm{loc}} = 1(8)$ for **MPP-mode** (hybrid modes)).

study up to 1024 processors (128 nodes) has been performed on the HITACHI SR8000 using a fixed workload per processor with matrix dimensions ranging from $D_{\mathrm{Mat}} = 2.62 \times 10^5$ to $D_{\mathrm{Mat}} = 2.68 \times 10^8$. For the hybrid modes we choose $n_k^{\mathrm{loc}} = 8$ and find in agreement with the inset of Fig. 7 an out-performance of the **task-mode** compared to the **vector-mode** of roughly 10 %. Considering the parallel efficiency (cf. inset of Fig. 8), the scalability problems in the **MPP-mode** as discussed in Sect. 4.2. are apparent for `AH matrices` also. Most notably the high scalability of the **task-mode** on hundreds of processors enables us the exceed the 100 GFlop/s barrier for sparse MVM algorithms on the HITACHI SR8000.

5 Conclusions

We presented a scalable sparse MVM algorithm based on a JDS format. Our parallel implementation was shown to guarantee an efficient use of modern vector computers (NEC SX5e) and to achieve high scalability on SGI Origin3800 as well as CRAY T3E systems. However, the main focus was on two "hybrid programming model" implementations optimized for SMP based supercomputers. As an example for an SMP system the HITACHI SR8000 TeraFlop/s system at LRZ Munich was chosen. We demonstrated that the "hybrid programming model" generally outperforms the MPP model for the matrices used in our work. Furthermore it was substantiated that an explicit thread programming technique simulating an asynchronous data transfer achieves best performance for MVM problems with a reasonable ratio of communication to computation. Besides the hybrid features with respect to architecture and programming model, another hybrid characteristic was clearly demonstrated for the HITACHI SR8000: Although based on RISC technology one SMP node can exceed the performance of comparable present-day vector processors (NEC SX5e) for vectorizable loops even if a vector-gather operation is involved. Combining hybrid features of the HITACHI SR8000 with "hybrid programming model" is the method of choice to achieve high performance for large scale sparse MVM algorithms.

Acknowledgements

The authors would like to thank M. Brehm, U. Küster, T. Lanfear, R. Rabenseifner and R. Wolff for helpful discussions. Special thanks go to F. Deserno, M. Kinateder and F. Wagner. The performance studies were performed at TU Dresden, NIC Jülich, LRZ Munich and HLR Stuttgart. The research was granted by the *Bavarian Network for High Performance Computing* (KONWIHR).

References

1. S. W. Bova *et al.*, The International Journal of High Performance Computing Applications, **14**, pp. 49-60, 2000.
2. L. A. Smith and P. Kent, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 6-9.
3. D. S. Henty, *Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modelling.* In Proceedings of SC2000, 2000.
4. H. Shan *et al.*, *A Comparison of Three Programming Models for Adaptive Applications on the Origin2000.* In Proceedings of SC2000, 2000.
5. W. D. Gropp *et al.*, *Performance Modeling and Tuning of an Unstructured Mesh CF Application.* In Proceedings of SC2000, 2000.
6. R. Rabenseifner, *Communication Bandwidth of Parallel Programming Models on Hybrid Architectures.* To be published in the proceedings of WOMPEI 2002, Kansai Science City, Japan. LNCS 2327.
7. Available at `http://www.top500.org`.
8. J. Dongarra *et al.*, *Iterative Solver Benchmark*, available at `http://www.netlib.org/benchmark/sparsebench/`.
9. R. Barrett *et al.*, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia (1994).
10. M. Kinateder *et al.*, E. Krause and W. Jäger, eds.: High Performance Computing in Science and Engineering 2000, Springer, Berlin (2001), pp. 188–204.
11. W. Schönauer, *Architecture and Use of Shared and Distributed Memory Parallel Computers*, eds.: W. Schönauer, ISBN 3-00-005484-7.
12. P. W. Anderson, Phys. Rev. B **109**, 1492 (1958).
13. Available at `http://www-fp.mcs.anl.gov/petsc`.
14. G. Wellein *et al.*, *Exact Diagonalization of Large Sparse Matrices: A Challenge for Modern Supercomputers*, In Proceedings of CUG SUMMIT 2001, CD-ROM.
15. M. Brehm, LRZ Munich, private communication.