On Reducing I/O Overheads in Large-Scale Invariant Subspace Projections

Hasan Metin Aktulga[1], Chao Yang[1], Ümit V. Çatalyürek[2], Pieter Maris[3], James P. Vary[3], and Esmond G. Ng[1]

[1] Lawrence Berkeley National Laboratory, Berkeley CA 94720, USA
[2] The Ohio State University, Columbus OH 43210, USA
[3] Iowa State University, Ames IA 50011, USA

Abstract. Obtaining highly accurate predictions on properties of light atomic nuclei using the Configuration Interaction (CI) method requires computing the lowest eigenvalues and associated eigenvectors of a large many-body nuclear Hamiltonian, H. One particular approach, the J-scheme, requires the projection of the H matrix into an invariant subspace. Since the matrices can be very large, enormous computing power is needed while significant stresses are put on the memory and I/O subsystems. By exploiting the inherent localities in the problem and making use of the MPI one-sided communication routines backed by RDMA operations available in the new parallel architectures, we show that it is possible to reduce the I/O overheads drastically for large problems. This is demonstrated in the subspace projection phase of J-scheme calculations on ^{6}Li nucleus, where our new implementation based on one-sided MPI communications outperforms the previous I/O based implementation by almost a factor of 10.

1 Introduction

The direct solution of the quantum many-body problem transcends several areas of physics and chemistry. Nuclear physics faces the multiple hurdles of a very strong interaction, three-nucleon interactions, and complicated collective motion dynamics. The configuration interaction (CI) method requires computing the many-body wavefunctions associated with the discrete energy levels of nuclei by partially diagonalizing the nuclear many-body Hamiltonian, H, in a many-body basis space constructed from harmonic oscillator single-particle wavefunctions [1]. Typically, one is only interested in a limited number of low energy states [2,3], but for certain applications, computing a relatively large number of states (and their wavefunctions) with a prescribed total angular momentum J is crucial. We will refer to this type of calculation as a *total-J calculation* throughout this paper. Investigating nuclear level densities as a function of J and excitation energy, and evaluating scattering amplitudes for different values of J [4,5] are among the target applications for total-J calculations.

In the total-J approach, eigenvalues and eigenvectors associated with a given J value are computed through a diagonalization of the total angular momentum

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 305–314, 2012.

square operator $\hat{J}^2$ [6], followed by a diagonalization of the projection of H onto a desired (and common) invariant subspace of H and $\hat{J}^2$. There are three major stages in this approach:

1. Computing the invariant subspace of $\hat{J}^2$ for a given eigenvalue $\lambda = J(J+1)$,
2. Projecting the H matrix into this subspace,
3. Finally, extracting the desired spectral information from the resulting lower dimensional Hamiltonian.

In [6], we present a multi-level method based on a greedy load-balancing algorithm to compute the invariant subspace Z of $\hat{J}^2$ efficiently on a large-scale distributed memory machine. Here we tackle the second stage of the total-J calculation, namely the projection of the H matrix into the subspace spanned by the columns of Z, i.e., $H' = Z^T H Z$.

Subspace projection calculation consists of two successive matrix multiplications, where the many-body Hamiltonian H is a square sparse symmetric matrix and Z has a block diagonal structure. These special properties of the matrices involved allow the subspace projection task to be divided into many independent subproblems of smaller sizes (see Sect. 2). However, due to the large dimensions of H and Z, such calculations demand significant amounts of computational resources, especially in terms of storage spaces. For example, in the problems that we study in Sect. 4, the dimensions of the H matrix becomes as large as 1.7×10^8 and the number of columns of Z is approximately 2.7×10^7.

In this paper, we describe an efficient scheme for performing large-scale invariant subspace projections on distributed memory machines. To exploit parallelism, we decompose the projection calculation into a number of smaller computational tasks, each of which can be completed by a single processing unit, as described in Sect. 3. A major decision that we must make is where to store the Z matrix, whose size can be on the order of terabytes, and how to access its diagonal blocks efficiently. We discuss and compare two strategies for storing Z. In the first scheme, which we describe in Sect. 4, Z is stored on the disk. Each processing unit reads in the required blocks of Z whenever they are needed. We will refer to this scheme as the out-of-core (OOC) implementation. Clearly, the OOC implementation is prone to severe I/O overheads (see Sect. 4). Our alternative scheme described in Sect. 5 is based on distributing the diagonal blocks of Z among all processors and fetching data from potentially remote memory by efficient one-sided MPI calls. However, our incore implementation suffers from another kind of overhead: communication latency, which can easily be overcome by buffering. Our experiments and observations are summarized in Sect. 5.

2 The Strutures of H and Z Matrices

Figure 1 illustrates the structures of matrices H and Z. Each row (and column) of H corresponds to, what is known in nuclear configuration interaction calculation as, a many-body basis state, which is a Slater determinant of single-particle states (anti-symmetrized product of single-particle states). The total number of

many-body states, D, is determined by the number of particles in the nucleus, n_{part}, and a truncation parameter, $N_{\max}$. A higher $N_{\max}$ value yields a more accurate finite dimensional approximation to the nuclear many-body Hamiltonian at the expense of an exponential growth in its dimension. As shown in Fig. 1, H is sparse. Its sparsity pattern is determined by the type of interaction used: a 3-body potential leads to a less sparse matrix than a 2-body potential. We use a 2-body interaction potential for the simulations presented in this paper, which means an entry H_{ij} of the Hamiltonian is non-zero only when the number of different single-particle states corresponding to row i and column j of H is at most 2.

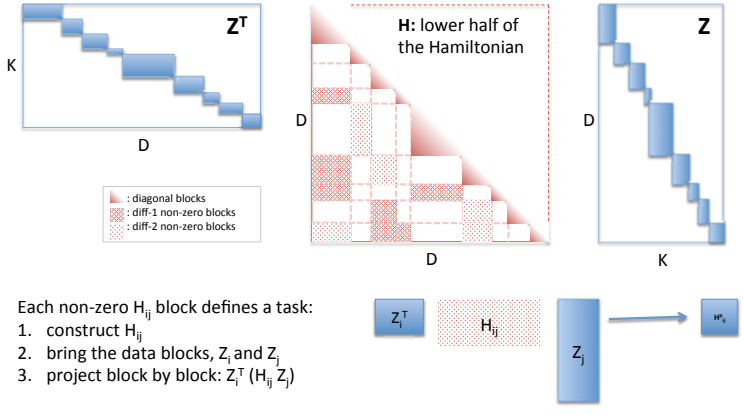

Fig. 1. Overview of the invariant subspace projection problem

The block structure of H seen in Fig. 1 results from a particular grouping of the many-body basis states based on their single particle quantum numbers (see [6] for details). Each non-zero block in H is itself sparse, and thus can be stored in a sparse matrix format. What is worth noting here is that each group of many-body basis state is invariant under the $\hat{J}^2$ operator. Consequently, such a grouping scheme produces a block diagonal representation of the $\hat{J}^2$ operator.

3 Task Decomposition for Parallel Processing

The special structures of H and Z matrices allow us to decompose the projection calculation into smaller tasks. Let i and j denote the block indices, then each non-zero H_{ij} block defines a subtask for the invariant subspace projection problem: $H'_{ij} = {Z_i}^T H_{ij} Z_j$, a sparse matrix multiplied with two dense blocks, resulting in a dense block of smaller dimensions (see Fig. 1). Accomplishing each small task involves construction of the H_{ij} block based on the interaction between the ith and jth many-body state groups and bringing the Z_i and Z_j blocks into local memory.

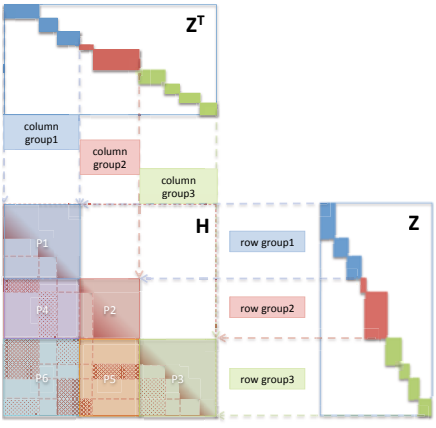

Fig. 2. Decomposition of the invariant subspace projection problem

Because H is symmetric, we configure processors into an $n_r \times n_r$ lower triangular grid T as shown in Fig. 2, so that the lower triangular part of the matrix can be distributed among $n_r(n_r+1)/2$ processors. The partitioning of H is based on a cyclic distribution of the many-body basis groups over the diagonal processors of T. This partitioning also yields a logical partitioning of the Z blocks among the diagonal processors. Row and column processor groups are created within T to facilitate data communication. The partitioned Z blocks are to be shared (logically) among all processors within the same row and column groups. Such a distribution scheme maximizes the data locality during the subspace projection calculations.

Directly associated with the partitioning of the H matrix is the distribution of the subspace projection subtasks, $Z_i^T H_{ij} Z_j$. Each processor is responsible for computing the projection of non-zero H_{ij} blocks within its partition. In order to accomplish these subtasks, each off-diagonal processor needs only two sets of diagonal blocks of Z (one mapped to its row group, the other one mapped to its column group), while a diagonal processor needs only a single set of Z blocks. We will refer to the blocks of Z mapped to a processor's row group as that processor's row data blocks, similarly to the blocks of Z mapped to a processor's column group as its column data blocks.

4 An Out-of-Core Approach

A $Z_i^T H_{ij} Z_j$ subtask has three steps: (i) construction of the non-zero H_{ij} based on the interaction between the many-body groups i and j, (ii) bringing the associated row data block Z_i and column data block Z_j into local memory, (iii) and finally computing ${Z_i}^T H_{ij} Z_j$. Ideally, we would like to store all blocks of Z

that will be needed by a processor to perform all its projection subtasks in its local memory. However, for large problems, this is generally not possible due to the limited amount of memory on each processor. Therefore, it is important to carefully consider where to store the diagonal blocks of Z and how to bring them to local memory.

One approach we examined is to store the diagonal blocks of Z on the disk and let each processor read them from there whenever necessary. Algorithm 1 gives the pseudo-code for our out-of-core approach, which we will refer to as the OOC implementation. In order to reduce read contention on a single file, n_r files (one file per diagonal processor) are created to store the diagonal blocks of the Z matrix. To reduce I/O overheads in OOC, once a column data block Z_j is read from the disk, all subtasks associated with that data block are processed, *i.e.*, we follow a column-major order for processing non-zero blocks of H.

```
input  : Processor column and row group indices: mycol and myrow
output: Projection of the part of H assigned to the processor

Open colfile that contains Z_j's mapped to the mycolth diagonal processor;
Open rowfile that contains Z_i's mapped to the myrowth diagonal processor;

foreach Z_j ∈ colfile do
    Read Z_j from colfile;
    foreach Z_i ∈ rowfile do
        if H_ij ≠ 0 then
            Read Z_i from rowfile;
            Construct H_ij;
            H'_ij = Z_i^T H_ij Z_j;
        end
    end
end
```

Algorithm 1. An OOC algorithm for computing $H'_{ij} = Z_i^T H_{ij} Z_j$

We tested the performance of the OOC implementation on the Hopper system at the National Energy Research Scientific Computing Center (NERSC) [2]. Table 1 summarizes the performance of the OOC implementation on some of the real problems that we typically solve using the total-J code. OOC performs well when the size of the Z matrix is relatively small. We suspect that this is due to the availability of I/O buffers managed by the OS kernel, where the entire column and (more importantly) row files that contain the needed blocks of Z can be buffered effectively, when these files are small in size. However, as the Z matrix becomes larger, the efficiency of the OOC implementation drops sharply – the useful work done, which corresponds to the time spent when processors are not idle, for the $N_{\max}$=14, J=3 case is only about 1%.

[2] See `http://www.nersc.gov/users/computational-systems/hopper/`

Table 1. Performance of the OOC implementation on total-J calculations of ^{6}Li with various parameters. $|Z|$ denotes the size of the Z matrix, n_{dblks} is the number of Z (data) blocks, n_{tasks} is the number of non-zero blocks in H. For each calculation, the total execution time (in seconds) and the percentage of overhead (*i.e.*, I/O time) is given.

calculation	$\|Z\|$ (GB)	n_{dblks}	n_{tasks}	n_{p}	overhead (%)	total (s)
$N_{\max}$=12, J=0	2.7	2.5×10^5	1.12×10^8	946	45%	159
$N_{\max}$=12, J=1	7.5	2.5×10^5	1.36×10^8	946	47%	204
$N_{\max}$=12, J=2	10.6	2.5×10^5	1.52×10^8	946	63%	319
$N_{\max}$=12, J=3	11.3	2.5×10^5	1.60×10^8	946	69%	358
$N_{\max}$=12, J=4	10.0	2.5×10^5	1.60×10^8	946	83%	551
$N_{\max}$=14, J=3	67.1	7.4×10^5	7.04×10^8	10,011	99%	9200

5 Distributed In-Core Approach

An alternative to the OOC implementation is to distribute and store a single copy of the blocks of Z in the local memory available to the processors. In a sense, we utilize the global address space available to the compute nodes as "disk". Our goal is to be able to tackle problems much bigger than the ones presented in Tab. 1 (*e.g.*, ^{6}Li, $N_{\max}$=16, J=5 where we expect $|Z| \approx 1$ TB excluding any auxiliary data structures) and to develop an implementation that can withstand the current trend of decreasing memory space per processor ratio. However, such an approach would require each processor to fetch data from remote memory belonging to a different processor during the projection calculation. The communication overhead may increase the turn-around time significantly, if regular MPI send/receives are used for fetching data. This overhead can be reduced by making use of one-sided MPI communication routines. On Hopper, each node is equipped with an RDMA engine which can handle remote memory access requests without interrupting the computation performed on processors residing on that node. So we take advantage of the one-sided **MPI_Get** operations available with Cray's xt-mpich2 MPI library on Hopper.

Our incore implementation balances the memory load among processors by distributing the blocks of Z logically mapped to the ith diagonal processor cyclically among processors that belong to the ith row and column communication groups. Before starting the projection calculations based on this distribution, each processor reads its share of Z blocks from its row and column data files into the memory. Contrary to our expectations, the incore version did not produce any improvements in terms of performance for the test cases listed in Tab. 1. A relatively small calculation for ^{6}Li, $N_{\max}$=12, J=0 took over 400 seconds on 946 processors. This performance compares unfavorably to the 159 seconds required in the OOC implementation. A detailed performance analysis reveals that reading the blocks of Z takes only about 20 seconds. The wall clock time spent in waiting for the completion of **MPI_Get** calls is about 250 seconds on average (more than 60% of the total completion time).

We believe that this unexpectedly large communication overhead is largely due to network latency, rather than the inadequacy of network bandwidth. Because in the ^{6}Li, $N_{\max}$=12, J=0 case, individual Z blocks are not large in size, but there are over 100 million small tasks, each of which typically requires two different blocks of the Z matrix. Column major processing of tasks helps keeping the number of **MPI_Get** calls roughly equal to the number of tasks, which is still prohibitively high. MPI-2 specification [7] requires even read-only accesses to remote memory locations, which is the exclusive usage of MPI one-sided calls in our incore algorithm, to happen inside an epoch. The only way to start and end an epoch in MPI-2 without synchronization is to enclose remote memory accesses within a pair of **MPI_Win_lock** and **MPI_Win_unlock** calls. This locking–unlocking protocol incurs an overhead of 4α, where α denotes the network latency, for each **MPI_Get** call. As discussed by Gropp et. al. [8], it is possible to detect this special access pattern and reduce the latency overhead to 0 by combining the locking–unlocking phase with the **MPI_Get** call itself. But to the best of our knowledge, this optimization has not been included in Cray's xt-mpich2 implementation.

Algorithm 2 illustrates the final version of the incore implementation. To reduce the latency overheads associated with starting and ending epochs, all **MPI_Get** calls destined to the same remote memory address space are consolidated within a single epoch. Consequently, instead of looping over the column and row files as in the OOC algorithm, the incore algorithm loops over processor sub-groups which now store the needed Z blocks. Each processor maintains a list of destinations for the Z blocks it needs and issues **MPI_Get** calls to easily fetch them. Note that the **MPI_Getv** call in Alg. 2 is not actually an MPI-library call, it is a wrapper which initiates an epoch, issues a sequence of **MPI_Get** calls destined to a target processor, and terminates the epoch. So between the initial version of the incore algorithm described above and its final version given here, the number of **MPI_Get** calls stays the same. However, the number of epochs created by a processor is reduced from being roughly equal to the number of tasks assigned to it down to n_r^2. The effect of this reduction can be seen immediately in ^{6}Li, $N_{\max}$=12, J=0 calculations on 946 processors where the running time of the final incore algorithm is 250 seconds (down from 400 seconds) and the communication overhead is reduced to 40% (down from over 60%).

In Fig. 3, we compare the performance of the OOC and the final distributed incore implementations for different J values in ^{6}Li, $N_{\max}$=12 calculations on 946 processors. While the OOC version initially outperforms the incore implementation due to the small size of the Z matrix, the incore implementation delivers up to 2.8x speed-up over the OOC version for larger values of J where the size of the Z matrix is considerably larger (see Tab. 1). But the real advantage of the incore implementation becomes evident during the much larger ^{6}Li, $N_{\max}$=14, J=3 calculations, where we obtain almost 10x speed-up over the OOC implementation, see Fig. 4.

```
input  : column & row group ids: mycol, myrow
input  : column & row group communicators: row_comm, col_comm
input  : manybody_groups (non-empty only on diagonal processors)
output: Projection of the part of H assigned to the processor

if diagonal processor then
    ids ← pids ∈ row_comm ∪ pids ∈ col_comm;
    host ← distribute manybody_groups cyclically over ids;
end
host ← Bcast(host, ids);
my_mb_groups ← Scatter(manybody_groups, ids);

cpids ← Bcast(ids, col_comm);      // processor subgroup to look for Z_i s
rpids ← Bcast(ids, row_comm);      // processor subgroup to look for Z_j s
my_dblks ← Load from column data file based on my_mb_groups;
my_dblks ← Load from row data file based on my_mb_groups;

foreach c ∈ cpids do
    cdlist ← {Z_i | need(Z_i) ∧ host(Z_i) = c};
    MPI_Getv(c, cdlist, cdblks);
    foreach r ∈ rpids do
        rdlist ← {Z_j | need(Z_j) ∧ host(Z_j) = r};
        MPI_Getv(r, rdlist, rdblks);
        foreach Z_i ∈ cdblks, Z_j ∈ rdblks do
            Construct H_ij;
            H'_ij = Z_i^T H_ij Z_j;
        end
    end
end
```

Algorithm 2. Pseudo-code for the incore implementation

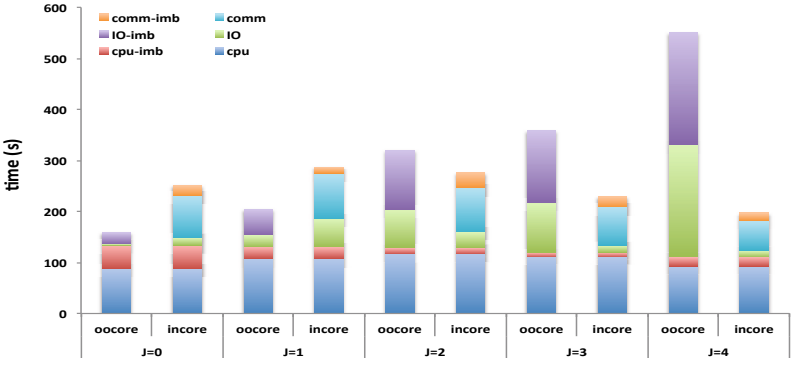

Fig. 3. Comparison of the performances of the incore and the OOC implementations. Total computation time (as denoted by the height of each bar) is examined in 6 parts: average CPU time (cpu), cpu load imbalance (cpu-imb), IO time (IO), IO time imbalance (IO-imb), communication time (comm), communication time imbalance (comm-imb). Note that, the OOC implementation does not do any communication. In the incore implementation, no IO-imb is measured.

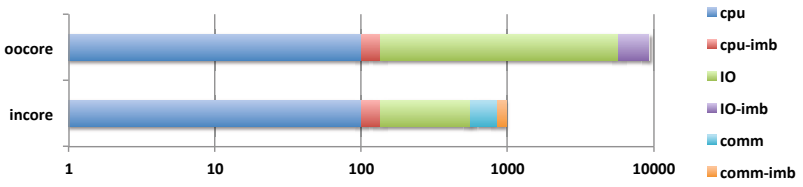

Fig. 4. Comparison of the performances of incore and OOC implementations for the ^{6}Li, $N_{\max}$=14, J=3 calculations on 10,011 processors, showing a 10x speed-up

6 Future Challenges

A detailed examination of the performance of incore algorithm reveals that the incore implementation is still prone to severe overheads. For example, for the $N_{\max}$=14, J=3 calculations on 10,011 processors, the percentage of useful computation (as measured by the ratio of average CPU time per processors to total wallclock time) is merely 10%. Out of the total 970 s, on average about 400 s is spent for reading the diagonal blocks of Z to memory and about 300 s is spent on communication. Another source of the inefficiency in the incore algorithm is the potential load imbalances among processors. Load imbalance caused by the basic round-robin distribution of tasks is not severe for ^{6}Li calculations where variations among task sizes are not drastic. But for heavier nuclei, we observe that there is a large variation among task sizes which could potentially cause severe load imbalances.

Some communication overheads may be reduced through further optimizations on Alg. 2. For example, one can pack multiple **MPI_Get** calls that are made to retrieve data from consecutive memory locations into one. Other one-sided communication libraries (such as ARMCI [10]) remains to be explored as they offer optimizations not present in the current MPI-2 implementations.

However, in order to alleviate the problems described above and further reduce overheads that were encountered in the subspace projection phase of total-J calculations, smarter heuristics, such as the technique presented in [9], which try to balance the task and memory load while minimizing communication overheads are necessary.

7 Conclusions

In this paper, we explore different approaches to tackle the challenging problem of large-scale invariant subspace projection problem arising in the context of total-J calculations and analyze their performances on real problems of interest to the nuclear physics community. We show that, by exploiting the inherent localities in the problem and making use of the MPI one-sided communication routines, it is possible to reduce the I/O and communication overheads drastically for large-scale data dependent problems. Despite the significant speed-ups achieved with the final version of the total-J code, our analysis shows that the

useful work to total execution time ratio is still low (as low as 10%). We identify the sources of inefficiencies through a careful examination of the performance profiles of individual processors and lay out future research directions that may reduce overhead and provide better scalability to this important problem.

We realize that eigenvalue calculations are central to several problems in the area of computational science and engineering. Also high performance data-intensive computing is a field with growing interest. Therefore we believe that the results of this work and insights we have gained can be of much broader interest.

Acknowledgment. The computational results were obtained at the National Energy Research Scientific Computing Center (NERSC), which is supported under contract number DE-AC02-05CH11232. Research was supported in part by the DOE grants DE-FC02-09ER41582 (SciDAC-UNEDF), DE-FG02-87ER40371 and DE-FC02-06ER2775; by the NSF grants CNS-0643969, OCI-0904809, OCI-0904802 and NSF-0904782.

References

1. Vary, J.P., Maris, P., Ng, E., Yang, C., Sosonkina, M.: Ab initio nuclear structure: The Large sparse matrix eigenvalue problem. J. Phys. Conf. Ser. 180, 012083 (2009)
2. Maris, P., Shirokov, A.M., Vary, J.P.: Ab initio nuclear structure simulations: The Speculative F-14 nucleus. Phys. Rev. C 81, 021301 (2010)
3. Maris, P., Vary, J.P., Navratil, P., Ormand, W.E., Nam, H., Dean, D.J.: Origin of the anomalous long lifetime of 14C. Phys. Rev. Lett. 106, 202502 (2011)
4. Shirokov, A.M., Mazur, A.I., Zaytsev, S.A., Vary, J.P., Weber, T.A.: Nucleon nucleon interaction in the J matrix inverse scattering approach and few nucleon systems. Phys. Rev. C 70, 044005 (2004)
5. Shirokov, A.M., Mazur, A.I., Vary, J.P., Mazur, E.A.: Inverse scattering J-matrix approach to nucleon-nucleus scattering and the shell model. Phys. Rev. C 79, 014610 (2009)
6. Aktulga, H.M., Yang, C., Ng, E., Maris, P., Vary, J.P.: Large-scale Parallel null space calculation for nuclear configuration interaction. In: Proc. of HPCS 2011, Istanbul, Turkey, July 4 - 8 (2011)
7. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI – The Complete Reference. The MPI-2 Extensions, vol. 2. MIT Press, Cambridge (1998)
8. Gropp, W., Thakur, R.: An Evaluation of Implementation Options for MPI One-Sided Communication. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 415–424. Springer, Heidelberg (2005)
9. Çatalyürek, Ü.V., Kaya, K., Uçar, B.: Integrated Data Placement and Task Assignment for Scientific Workflows in Clouds. In: Proc. of HPDC, The Fourth International Workshop on Data Intensive Distributed Computing (DIDC) (June 2011)
10. Nieplocha, J., Tipparaju, V., Krishnan, M., Panda, D.: High performance remote memory access comunications: The ARMCI approach. International Journal of High Performance Computing and Applications 20(2), 233–253 (2006)