

An Extensible Global Address Space Framework with Decoupled Task and Data Abstractions

Sriram Krishnamoorthy¹ Umit Catalyurek² Jarek Nieplocha³ Atanas Rountev¹
P. Sadayappan¹

¹ Dept. of Computer Science and Engineering, ² Dept. of Biomedical Informatics
The Ohio State University

³ Pacific Northwest National Laboratory

Abstract

Although message passing using MPI is the dominant model for parallel programming today, the significant effort required to develop high-performance MPI applications has prompted the development of several parallel programming models that are more convenient. Programming models such as Co-Array Fortran, Global Arrays, Titanium, and UPC provide a more convenient global view of the data, but face significant challenges in delivering high performance over a range of applications. It is particularly challenging to achieve high performance using global-address-space languages for unstructured applications with irregular data structures.

In this paper, we describe a global-address-space parallel programming framework with decoupled task and data abstractions. The framework centers around the use of task pools, where tasks specify operands in a distributed, globally addressable pool of data chunks. The data chunks can be addressed in a logical multi-dimensional “tuple” space, and are distributed among the nodes of the system. Locality-aware load balancing of tasks in the task pool is achieved through judicious mapping via hyper-graph partitioning, as well as dynamic task/data migration. The framework implements a transparent interface for out-of-core data, so that explicit orchestration of movement of data between disks and memory is not required of the programmer. The use of the framework for implementation of parallel block-sparse tensor computations in the context of a quantum chemistry application is illustrated.

1 Introduction

Computers have made dramatic strides in speed over the last two decades, but unfortunately the difficulty of programming parallel computers has not eased. As computers

have increased in achievable performance, making it feasible to accurately model more and more complex phenomena, the time and effort required to develop the software has become the bottleneck in many areas of science and engineering.

From a programmer’s viewpoint, the complexity of the code required to implement a given algorithm or simulation is a function of the level of detail the programming model exposes to the programmer, together with level of detail required to manage performance-related aspects of the underlying hardware (such as memory hierarchy, processor association, etc.) The higher the level of abstraction of the software environment and the more compiler/runtime handle these details, the more productive the programmer can be in terms of developing code.

The dominant approach at present is message passing using MPI, which requires the programmer to explicitly partition the work, map it onto the parallel computer, and schedule its execution. Data movement between processes must also be managed by the programmer, by the sending of messages which require explicit coordination of sender and receiver. This approach makes parallel computing very tedious and error-prone because of the myriad low-level details the programmer must contend with. One of the reasons for the continuing use of the MPI programming model is that by leaving all of the details up to the programmer, it is possible to obtain high performance — if the programmer puts in the effort to do so. But as the architecture of parallel computer systems gets more complex, the explicit orchestration of computation and communication for optimum performance will get increasingly difficult.

At the other end of the spectrum from the MPI approach would be a programming environment that would require the programmer to specify only the algorithm, with the compiler/runtime environment managing all issues associated with parallelism. Relieving the programmer of having to worry about the details of the parallel decomposi-

tion and data movement raises the level of abstraction at which they are able to think about the problem and increases their productivity. Such an approach would require a compiler/runtime system of great sophistication in order to realize high performance. In general, completely automatic transformation of a non-explicitly-parallel specification for efficient parallel execution may be intractable. However, for important subclasses of computation involving regular access on dense matrices, sophisticated frameworks [2, 3, 1] have been developed for analysis and transformation.

Intermediate between these two extremes is the emerging class of programming models that require explicit specification of partitioning and scheduling of computation onto processes, but offer a global-shared view of data, such as UPC [29], Co-Array Fortran [7], Titanium [31, 27], and Global Arrays [20, 21]. These models are generally considered easier to program than MPI because they involve only one-sided communication for data transfer, which is in some cases implicit in the program syntax, together with their global-shared view of the program data. However, they still require explicit specification of the parallelism. These approaches have become prominent only recently, but are drawing a great deal of interest as alternatives to MPI as the basic parallel programming model in a variety of areas. For example, essentially all scalable parallel quantum chemistry packages use either Global Arrays or an equivalent [12, 30, 16, 17, 24, 10].

For computations involving data structures more general than dense matrices, performing completely automatic program transformations to parallelize them and optimize their execution time is a challenging task. Extant parallel programming models, while simplifying the programming of such computations, might require a lot of effort to achieve comparable performance. We propose an intermediate model in which the data and computation abstractions are geared towards expressing and exploiting the locality inherent in the problem. The data abstraction encapsulates the locality of access. The user expresses the parallelism in the computation in the form of a *task pool*, which is then used by the runtime system to schedule the computation to maximize locality. The decoupling of the computation and data abstractions enables the implementation of efficient operations on existing data structures, extending their capabilities. In addition, the computation abstractions can be leveraged to quickly develop new libraries for domain-specific data structures.

In this paper, we present our experiences with this approach in the context of the Tensor Contraction Engine (TCE) synthesis system. The computation is characterized by operations on generalized multi-dimensional block-sparse arrays. The motivating applications are presented in Section 2. The abstraction for block-sparse arrays is discussed in Section 3. The task pool abstraction is presented

with an illustration in Section 4. Section 5 discusses the opportunities for optimization using the abstractions discussed, in the context of the problem domain. The implementation framework is discussed in Section 6. Section 7 explores the related work, and Section 8 concludes the paper.

2 Motivation

One of the major motivations for the development of the proposed primitives is our work on the Tensor Contraction Engine (TCE) [4] synthesis system. TCE is a domain-specific compiler for expressing ab initio quantum chemistry models. The TCE takes as input a high-level specification of a computation, expressed as a set of tensor contraction expressions, and transforms it into efficient parallel code. Each tensor contraction expression is comprised of a collection of multi-dimensional summations of products of several block-sparse input arrays. An operation on the indices of the segments that form a block of an array determines if that block is non-zero. The wide-ranging sizes of the blocks lead to significant variations in the computation and communication times involved in processing a block. The large sizes of the arrays can significantly increase communication costs, if locality is not taken into account.

In general, applications with the following characteristics can benefit from our proposed scheme:

- Can be partitioned into independent tasks,
- Involve many more tasks than the number of processors,
- Have wide variation in task execution times, and
- Operate on coarse-grain data, and incur communication costs if the task and the data it operates on are not co-located.

Note that computations with data dependences can also benefit from this mechanism, provided there is enough parallelism at any point in the computation. For example, while performing a sequence of block-sparse matrix multiplies, each matrix multiply can be treated as a set of independent tasks and processed using this mechanism.

3 Abstraction for Block-Sparse Matrices

In this section we detail our data abstraction for block-sparse matrices. The abstraction is shown in Figure 1. For brevity, we use a pseudo-code notation; the actual API is implemented in C/C++. The data abstraction provides collective functions for creating and destroying arrays and non-collective functions to get/put data from/to the distributed block-sparse array.

```

Types::
BsaIndex      !Index to block-sparse array
BsaObject     !Block-sparse array object

Function Parameters::
BsaObject obj           !Handle to block-sparse array
BsaIndex ind           !Handle to index to the array
Integer ndims          !Number of dims of the array
Integer nblocks[ndims] !Num blocks along each dimension
Integer blocks[ndims][nblocks] !The size of each block segment
Integer brick[ndims]   !Brick size along each dimension
void *bitmap           !Bitmap specifying non-zero blocks
Fn_t isNonZero         !Function. Arguments: block indices,
                       !Return Value: true if non-zero block

Integer *brick         !Size of brick along each dim
Integer *brickIndex[ndims] !Index of a brick
void *buf              !Local buffer
Integer dim            !Dimension referenced

Functions::
!Create an index -- collective function
BsaIndex bsaCreateIndex(ndims, nblocks, blocks, Fn_t isNonZero, brick);
BsaIndex bsaCreateIndex(ndims, nblocks, blocks, bitmap, brick);

!Create block-sparse array -- collective function
BsaObject bsaCreateArray(ind);

!Retrieve a brick -- one-sided
void bsaGetBrick(obj, brickIndex, buf);

!Store a brick -- one-sided
void bsaPutBrick(obj, brickIndex, buf);

!Atomically update a brick -- one-sided
void bsaUpdateBrick(obj, brickIndex, buf);

!Check is a brick is non-zero - one-sided
bool bsaIsBrickNonZero(obj, brickIndex);

!Enquire #bricks along a dimension
Integer bsaGetNBricksAlongDim(obj, dim);

!Destroy block-sparse array
void bsaDestroyArray(obj);

!Destroy index
void bsaDestroyIndex(ind);

```

Figure 1. Abstraction for multi-dimensional block-sparse arrays

A brick size is specified while creating a block-sparse array. Alternatively, the user can specify the typical access pattern, to provide hints on the choice of appropriate brick sizes. The non-zero blocks of the array are divided into bricks of this size, which are then distributed amongst the processors in a round-robin fashion. This ensures a uniform distribution of the data among all processors. A small brick size allows for a more uniform distribution of the data amongst the processors. On the other hand, a large brick size allows for coarse-grained, and possibly more efficient, computation and potential reduction in the communication cost, due to amortization of the communication latency.

The process of creating an array is divided into two steps. The index is created first, using functions `bsaCreateIndex` from Figure 1. This involves traversing the bricks in the array, and determining the distribution of the non-zero bricks amongst the processes. The array is then created through function `bsaCreateArray` using this index. The decoupling of the creation of the index from the actual array creation simplifies the construction of multiple aligned arrays using the same index structure. In computations in which memory is dynamically allocated and de-allocated, the index can be computed once, while the actual memory for the array is dynamically allocated and de-allocated.

The arrays can be created by specifying the number of dimensions, the number of blocks, and the actual block sizes. In addition, a bitmap can be provided to specify whether a block is zero. Alternatively, the programmer can provide a function that takes as argument the block indices and returns whether it is zero.

4 Computation Abstraction: Task Pool

The task-pool abstraction shown in Figure 2 enables the specification of a set of independent tasks to be executed in parallel. For each such set, all processes collectively create a `TaskPool` object using the `tpCreateTaskPool` function.

Each task in the task pool is identified by the routine to be invoked to process that task (accessed through a function handle) and the set of *locality elements* it operates upon. In addition, any private data specific to that task can also be specified. Each locality element corresponds to a global data brick, identified by the array object, a tuple corresponding to the brick index, and the access mode. Three access modes are supported: read, write, and accumulate access modes allow for put, get, and accumulate of global data, respectively.

Tasks are added to a task pool using the `tpAddTask` function. The creation and addition of tasks to the task pool is done by all the processes, in a replicated fashion. Once all the tasks have been added to the task pool, `tpSeal` is used to *seal* the work pool. This function is invoked once for a

```
Types::
TaskPool      !Handle to task pool
Fn_t          !Function to process a task
LocalityInfo  !Locality information
PrivateData   !Local information for a task

Function Parameters::
TaskPool      tpHandle   !Task pool
Fn_t          fn         !Processing function
Integer       nLocInfo   !# locality elements
LocalityInfo  *locs      !Locality elements
PrivateData   *pvt       !Local data

Functions::
TaskPool tpCreateTaskPool();
void      tpAddTask(tpHandle, fn,
                   nLocInfo, locs, pvt);
void      tpSeal(tpHandle);
void      tpProcess(tpHandle);
```

Figure 2. Abstraction to specify a set of independent tasks to be executed in parallel

task pool and is used to perform start-time optimizations.

Subsequently, all the processes collectively invoke `tpProcess` to process the tasks in the task pool. A task pool, once created, can be processed multiple times. The cost of start-time optimizations, performed once, are thus amortized.

The work-sharing construct is illustrated using an implementation of block-sparse matrix multiply, shown in Figures 3 and 4. The multiplication is of the form

$$C[i, j] += A[i, k] * B[k, j]$$

The brick sizes along the different dimensions are assumed to be defined elsewhere. Parameters `bsaA`, `bsaB`, and `bsaC` correspond to block-sparse arrays A , B , and C , respectively.

Figure 4 shows routine `BrickMatmul` used to process an individual task, matrix-multiply involving a brick from each of the arrays. Note that no explicit communication is involved. The routine assumes that all input data are read into local memory and all output data are written/accumulated into global memory. Figure 3 shows the implementation of parallel matrix multiply using this routine.

5 Opportunities for Optimization

The data abstraction for block-sparse matrices, together with the computation abstractions allow an explicit specification of the locality in the computation. This opens up rich avenues of optimization. The optimizations can increasingly reduce the level of detail required of the programmer.

```

!Brick sizes. Defined elsewhere
extern int bi, bj, bk;
Matmul(bsaC, bsaA, bsaB) ::
  TaskPool tpHandle
  localityInfo locs[3]
  nBricksI = bsaGetNBricksAlongDim(bsaC,0);
  nBricksJ = bsaGetNBricksAlongDim(bsaC,1);
  nBricksK = bsaGetNBricksAlongDim(bsaA,1);
  !Create task pool
  tpHandle = tpCreateTaskPool()
  for i=0 to nBricksI - 1
    for j=0 to nBricksJ - 1
      int cbrick[2] = {i,j}
      if bsaIsBrickNonZero(bsaC, cbrick)
        for k=0 to nBricksK - 1
          int abrick[2] = {i,k}
          int bbrick[2] = {k,j}
          if bsaIsBrickNonZero(bsaA, abrick) AND
            bsaIsBrickNonZero(bsaB, bbrick)
            !Brick sizes
            int pvt[3]={bi,bj,bk}
            localityInfo locs[3] = {
              (bsaA,abrick,ACCESS_READ)
              (bsaB,bbrick,ACCESS_READ)
              (bsaC,cbrick,ACCESS_UPDATE)
            };
            !add task to task pool
            tpAddTask(tpHandle, BrickMatmul,
              3, locs, pvt)
  !Any start-time optimizations
  tpSeal(tpHandle)
  for i = 0 to maxiter !Iterative computation
    !Process all tasks, every iteration
    tpProcess(tpHandle)
  tpDestroy(tpHandle) !Destroy task pool

```

Figure 3. Block-sparse matrix multiply. Each task is processed by BrickMatmul. The task pool is processed maxiter times, but is created and sealed once

Locality-aware Load Balancing The specification of the computation can be used to load balance the computation while ensuring locality. The data structure is distributed amongst the physical memories of the processors. The tasks can be scheduled amongst the processors to minimize communication. In addition, effective caching can further reduce communication.

Automatic Out-of-core Data Management The computation specification allows the data to reside on disk instead of the global memory. The computation is scheduled so as to minimize the I/O required. Effective computation scheduling can automate the movement of data between disk and global memory, thus greatly simplifying the programming of out-of-core computations. Different models of I/O can be used, such as collective or non-collective I/O. The choice is either based on user directives or an under-

```

void BrickMatmul(int nLocInfo,
  localityInfo *locs,
  void *buf[],
  privateData *pvt) ::
  integer Ni, Nj, Nk, i, j, k
  double *A, *B, *C

  !Actual communication is external
  !to this function
  !Fetch pointer to data/buffer
  A = buf[0]
  B = buf[1]
  C = buf[2]

  Ni = pvt[0] !Brick sizes
  Nj = pvt[1]
  Nk = pvt[2]

  !Matrix multiply for this task
  for i = 0 to Ni-1
    for j = 0 to Nj-1
      for k = 0 to Nk-1
        C[i,j] += A[i,k] * B[k,j]

```

Figure 4. Routine to process a single task in block-sparse matrix multiply

lying model of the execution costs. In both cases, the actual details of data movement are abstracted away from the programmer. Note that the in-memory data distribution of arrays can also be determined to optimize communication cost.

Global Optimizations Given a specification of sequences of data parallel tasks and their dependences, global optimizations can be performed. This could include reusing an intermediate in memory thus reducing disk storage and I/O. Note that the abstractions enable such optimizations without any additional input from the programmer.

6 Implementation Realization

The task pool abstraction and the block-sparse array data structure are being implemented using the ARMCI library [18]. ARMCI is a constituent of the Global Arrays programming suite [22]. The suite provides a set of interoperable programming models, each at a different level of abstraction.

The Aggregate Remote Memory Copy Interface (ARMCI) library [18] provides a distributed-memory view with one-sided access to remote data. It has a rich set of primitives for non-blocking operations, and contiguous and non-contiguous data transfers optimized to hide latency. ARMCI forms the underlying communication layer for a number of compile/runtime systems, including Co-Array

Fortran [8], GPShMEM [23], and Global Arrays. The next higher level is the Global Arrays (GA) library. GA exposes a global view of a dense multi-dimensional array distributed amongst the local memories of processors. It is similar to distributed shared-memory models in providing an explicit acquire-release protocol, but differs with respect to the level of explicit control in moving blocks of data in multidimensional arrays between remote global storage and local storage. The functionality provided by GA has proved useful in the development of large scale parallel quantum chemistry suites such as NWChem [12] (which contains over a million lines of code), adaptive mesh refinement codes such as NWPhys/NWGrid (www.emsl.pnl.gov/nwphys) and applications in other areas [22]. The Disk Resident Arrays (DRA) model [19] extends the GA programming model to secondary storage. The suite is also inter-operable with MPI. These varied levels of abstractions have been shown to achieve high performance, while being simpler to program.

The locality-aware load-balancing of tensor contractions involving block-sparse arrays in the global memory was demonstrated in [14]. We used a hypergraph-partitioning approach [15] to minimize communication while ensuring load balance. The approach was shown to perform better than conventional solutions to the problem.

We are currently investigating locality-aware load-balancing of tensor contractions involving block-sparse arrays on disk. An extension of the hypergraph partitioning approach is envisioned.

7 Related Work

Abstractions for block-sparse matrices exist in the context of linear algebra and iterative solvers [9]. Aztec [28] is a parallel iterative solver package that provides a global view of a distributed matrix. Advanced partitioning techniques [11] are used to determine the computation distribution and mapping. We provide a general-purpose abstraction for block-sparse matrices. The computation abstractions and the mechanisms for optimization are not tightly coupled with block-sparse matrices, and can be utilized in a wide range of contexts.

Dynamic load-balancing based on work-stealing has been studied, particularly for state-space search [26]. Charm++ [13] supports dynamic load-balancing by object migration. Cilk [25] supports load-balancing of computations based on work-stealing. OpenMP exploits parallelism at the loop level by distributing different iterations to different processors. Locality is not taken into consideration in any of these schemes.

Çatalyürek and Aykanat [5] have used hypergraph-partitioning to parallelize sparse matrix-vector multiplications. Chang et al. [6] performed parallel data aggregation

based on hypergraphs.

8 Conclusions

We presented an abstraction for computation and data that results in an extensible framework. While improving upon the extensibility and generality of existing approaches to library creation, our approach enables better optimization of key kernels than using typical parallel programming languages. Our experience with block-sparse matrices shows the feasibility of this approach. We plan to provide support for other data structures and evaluate the performance as compared to state-of-the-art libraries.

Acknowledgments

We thank the National Science Foundation for the support of this research through grants 0121676, 0403342, and 0509467, and the U.S. Department of Energy through award DE-AC05-00OR22725. We thank the Molecular Sciences Computing Facility (MSCF) at the Pacific Northwest National Laboratory (PNNL) and the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

References

- [1] V. Adve and J. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.
- [2] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proc. ACM Intl. Conf. on Supercomputing*, pages 141–152, 2000.
- [3] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loops nests. In *Proc. of SC 2000*, 2000.
- [4] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc. of Supercomputing 2002*, November 2002.
- [5] U. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse matrix-vector multiplication. *IEEE TPDS*, 10(7):673–693, 1999.
- [6] C. Chang, T. Kurc, A. Sussman, U. V. Çatalyürek, and J. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 2001.
- [7] Co-array fortran. <http://www.co-array.org/>.
- [8] C. Coarfa, Y. Dotsenko, and J. Mellor-Crummey. A Multi-Platform Co-Array Fortran Compiler. In *Proc. of PACT*, 2004.

- [9] I. S. Duff, M. Marrone, G. Radicati, and C. Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: a user-level interface. *ACM Trans. Math. Softw.*, 23(3):379–401, 1997.
- [10] M. Guest. Computing for science. <http://www.dl.ac.uk/CFS/cfs.html>, 8 March 2004.
- [11] B. Hendrickson and R. Leland. The Chaco user’s guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.
- [12] High Performance Computational Chemistry Group. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.6*. Pacific Northwest National Laboratory, 2004.
- [13] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA’93*, pages 91–108. ACM Press, September 1993.
- [14] S. Krishnamoorthy, J. Nieplocha, and P. Sadayappan. Data and computation abstractions for dynamic and irregular computations. In *Proc. 12th Annual International Conference on High Performance Computing (HiPC)*. Springer Verlag, 2005.
- [15] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
- [16] H. Lischka and T. Mueller. The columbus parallel CI program project. http://www.itc.univie.ac.at/~hans/Columbus/columbus_parallel.html, May 6 2004.
- [17] Molcas 6. <http://www.teokem.lu.se/molcas/>.
- [18] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP)*, 1999.
- [19] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-Of-Core Computations. In *Proc. 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204, 1996.
- [20] J. Nieplocha and R. Harrison. Shared-memory programming in metacomputing environments: The global array approach. *The Journal of Supercomputing*, 11:119–136, 1997.
- [21] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A portable shared memory model for distributed memory computers. In *Proc. Supercomputing ’94*, pages 340–349, 1994.
- [22] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Intern. J. High Perf. Comp. Applications*, to appear, 2005.
- [23] K. Parzyszek, J. Nieplocha, and R. A. Kendall. A Generalized Portable SHMEM Library for High Performance Computing. In *Proc. of the IASTED Parallel and Distributed Computing and Systems*, pages 401–406, November 2000.
- [24] Q-Chem, Inc. Q-chem. <http://www.q-chem.com/>, May 13 2004.
- [25] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.
- [26] A. Sinha and L. Kalé. A load balancing strategy for prioritized execution of tasks. In *Seventh International Parallel Processing Symposium*, pages 230–237, Newport Beach, CA., April 1993.
- [27] Titanium. <http://www.cs.berkeley.edu/projects/titanium/>.
- [28] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid. Official Aztec user’s guide: Version 2.1. Technical report, Sandia National Laboratories, 1999.
- [29] Unified parallel c. <http://upc.nersc.gov/> and <http://upc.gwu.edu/>.
- [30] H. J. Werner, P. J. Knowles, R. Lindh, M. Schütz, P. Celani, T. Korona, F. R. Manby, G. Rauhut, R. D. Amos, A. Bernhardsson, A. Berning, D. L. Cooper, M. J. O. Deegan, A. J. Dobbyn, F. Eckert, C. Hampel, G. Hetzer, A. W. Lloyd, S. J. McNicholas, W. Meyer, M. E. Mura, A. Nicklass, P. Palmieri, R. Pitzer, U. Schumann, H. Stoll, A. J. Stone, R. Tarroni, and T. Thorsteinsson. Molpro, version 2002.6, a package of ab initio programs. <http://www.molpro.net>, 2003.
- [31] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11–13), 1998.