A Fine-Granular Programming Scheme for Irregular Scientific Applications

Haowei Huang[1(✉)], Liehui Jiang[2], Weiyu Dong[1], Rui Chang[1], Yifan Hou[1], and Michael Gerndt[3]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, Henan, China
haowei.huang@yahoo.com

[2] China National Digital Switching System Engineering and Technological Research Center, Zhengzhou 450000, Henan, China

[3] Fakultaet Fuer Informatik, Technische Universitaet Muenchen, 85748 Garching Bei Muenchen, Germany

Abstract. HPC systems are widely used for accelerating calculation-intensive irregular applications, e.g., molecular dynamics (MD) simulations, astrophysics applications, and irregular grid applications. As the scalability and complexity of current HPC systems keeps growing, it is difficult to parallelize these applications in an efficient fashion due to irregular communication patterns, load imbalance issues, dynamic characteristics, and many more. This paper presents a fine granular programming scheme, on which programmers are able to implement parallel scientific applications in a fine granular and SPMD (single program multiple data) fashion. Different from current programming models starting from the global data structure, this programming scheme provides a high-level and object-oriented programming interface that supports writing applications by focusing on the finest granular elements and their interactions. Its implementation framework takes care of the implementation details e.g., the data partition, automatic EP aggregation, memory management, and data communication. The experimental results on SuperMUC show that the OOP implementations of multi-body and irregular applications have little overhead compared to the manual implementations using C++ with OpenMP or MPI. However, it improves the programming productivity in terms of the source code size, the coding method, and the implementation difficulty.

1 Introduction

HPC is currently experiencing very strong growth in all computing sectors. Many HPC systems are used for accelerating different kinds of calculation-intensive applications including quantum physics, weather forecasting, climate research, oil and gas exploration, molecular dynamics, and so on [1–4]. The major programming interfaces are OpenMP [5,6], MPI [7–9], and CUDA [10,11]. In addition, a large number of high-level programming models have been developed to improve the programming productivity and implementation efficiency as well, e.g., High Performance Fortran (HPF) [12,13], Charm++ [14–16], and Threading Building Blocks (TBB) [17,18]. All these high-level programming approaches are designed to obtain better programming productivity using higher level abstraction or automatic parallelization. However, it is still complicated for programmers to manage irregular scientific applications in an efficient and scalable fashion in terms of decomposing the computational domain, managing irregular communication patterns among processes, and manipulating data migration among processes, maintaining computational load balance, and so on. For example, a molecular dynamics (MD) simulation [19] is a form of N-body [20] computer simulation in which molecules interact with other molecules within a certain domain for a period of time. The molecules may move in the domain according to the interactions with others, which changes their storage layout and communication pattern during execution. In order to improve the performance of such irregular applications, researchers apply linked cells algorithms and bi-section decomposition method which needs runtime re-distribution.

Different from current programming models starting from the global data structure, we present a fine granular programming scheme for irregular scientific applications. It provides a programming interface that supports writing applications by focusing on the finest granular elements and their interactions. They are organized as an *Ensemble*, which manages the elements, topologies, and high-level operations. By using the high-level operations explicitly, developers can control the actions of the elements including communication, synchronization, and parallel operations. In this paper, we introduce an abstract machine model, programming interface, and implementation framework ported on different types of systems on SuperMUC [21].

2 Abstract Machine Model

As can be seen from Fig. 1, the machine model is an abstract architecture composed of a Control Processor (CP) and a large number of distributed Fine Granular Processors (FGPs). The major interactions between the CP and FGPs are described as follows:

1. *Explicit communication among FGPs*: It is triggered by the CP explicitly. Point-to-Point communication between FGPs is not supported due to low efficicency.
2. *Parallel computations of FGPs*: It starts the computation of FGPs in the form of parallel operations.
3. *Collective operations among FGP*: The CP is able to trigger collective operations on a set of FGPs. All the participating FGPs start the operations cooperatively to get collective results.
4. *Collective operations between CP and FGPs*: The CP can access the local memory of FGPs in the machine by explicit collective operations.

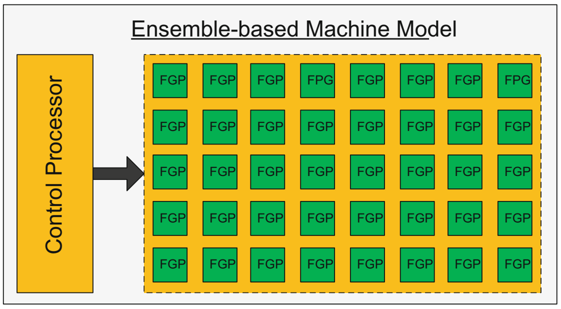


Fig. 1. Abstract machine model

3 Programming Interface

An object-oriented(OO) programming interface is designed on top of the abstract machine model. It consists of a template hierarchy starting from three top-level base templates *ElementaryPoint*, *Ensemble*, and *Topology*. These base templates have derived templates called application-specific templates, which support multi-body, irregular grid, and regular grid applications respectively. User-defined entities with local properties and operations can be defined as C++ classes derived from the application-specific templates. The organization is shown in Fig. 2.

Definition 1. *An ElementaryPoint(EP) is a software entity that represents the finest granular computational object in the domain of an application. The ensemble is a software container that stores a set of EPs and manages their local information, communication patterns, and computation. A topology defines a communication pattern resulting from the need for the information of a set of EPs in the ensemble. The EPs can exchange their status based on certain topologies.*

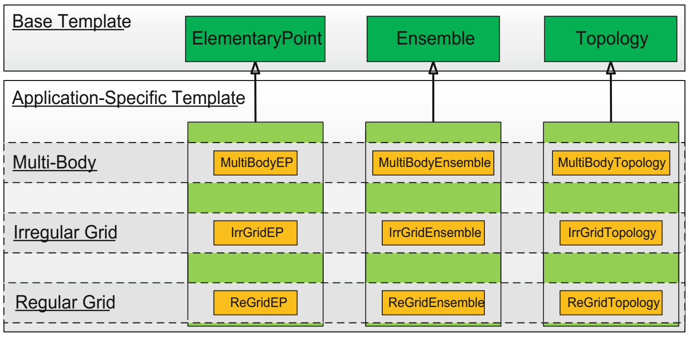


Fig. 2. Organization of the template hierarchy

The base templates for creating elementary points are *ElementaryPoint* and its derived templates *MultiBodyEP*, *IrrGridEP*, and *ReGridEP*. *Ensemble* and its derived templates are used to create the ensemble of an application. The major high-level operations in *Ensemble* are described as follows:

1. *SC-update*: It supports exchanging the complete information as well as partial information of EPs in the ensemble based on a certain topology.
2. *parallel*: It triggers member functions of EPs to execute in parallel. The template parameter is a function object adapted from a member function of *ElementaryPoint*.
3. *collective*: The *collective* operation currently consists of *allReduceOp* and *reduceOp*. *allReduceOp* provides collective reduction operations that return the result in all the involved EPs.
4. *getNghbList*: This operation is called by an individual EP in order to get a list of its neighboring EPs from the ensemble based on a topology. After it is accomplished, the EP can access data in the neighbor list for local computations.

Topology is used to create topologies, which keep the communication patterns of EPs. The major operations of *Topology* are shown as follows:

1. *initialization*: It initialize the internal data structures of *Topology*.
2. *createNeighborList*: If the topology is the root topology, it creates the neighbor EP list for the EPs.
3. *updateTopology*: It rebuilds a new topology according to the runtime information or the information specified by the users.

4 Implementation Framework

The implementation framework consists of machine-specific libraries including a sequential library, an OpenMP-based library, and an MPI-based library ported on SuperMUC. It is currently designed for multi-body and irregular grid applications. A single ensemble-based program can be compiled and linked to different executables by these libraries.

4.1 OpenMP-Based Library

It aggregates the computation of a group of EPs and binds it to a single thread. On NUMAs, all the EPs are initially stored in the physical memory of the socket running the master thread. It is not efficient that the threads residing on other sockets have to access the EPs by non-local memory accesses. Therefore, we apply a reallocation and re-indexing strategy to distribute EPs across different physical memory of the sockets.

Reallocation and Re-Indexing to Manage EPs and Their Shadow Copies. The OpenMP-based library integrates METIS [22] library to distribute the EPs across physically distributed memory. The *indirection* array is generated from the output of METIS. It gives the information of the thread affinity of all the EPs. The size of the array is the size of *EP_Set* called *numEP*. The *numEP/numSocket* elements of the *indirection* array keeps the identifiers of EPs stored in the physically memory of the sockets sequentially.The *indirection* array is organized in such a way. The the first *numEP/numSocket* elements of the *indirection* array (indices from 0 to $numEP/numSocket - 1$) stores the identifiers of EPs stored in the physically memory of socket#0. The second *numEP/numSocket* elements (indices from *numEP/numSocket* to $2 * numEP/numSocket - 1$) stores the identifiers of EPs stored in the physical memory of socket#1, and so on and so forth. The pseudo code of the EP reallocation is shown in Algorithm 1.

Algorithm 1. The EP reallocation

```
void reallocation(){
     EP *EP_Set = (EP*) malloc(numOfEPs * sizeof(EP));
#pragma omp parallel for
     for(i=0;i<numOfEPs;i++)
          EP_Set[i] = Buf_EP[indirection[i]];
     free(Buf_EP);
}
```

In order to avoid frequent accesses to the *indirection* array, we create *indexOrigin2New* and *indexNew2Origin* arrays to manage the re-indexing translation. The *indexOrigin2New* array is used for the translation from orignial indices to new indices, while *indexNew2Origin* is used for the translation from the new indices back to orignial indices. Both *indexOrigin2New* and *indexNew2Origin* are organized in such a way. The *indexNew2Origin* array and the *indirection* array described above have the same organization. The *indexOrigin2New* array is generated from *indexNew2Origin* according to the rule:

$$indexOrigin2New[indexNew2Origin[i]] = i;$$

For example, 8 EPs are resided on socket#0 and socket#1, the partitioning result generated from METIS is $[0, 1, 0, 1, 1, 0, 0, 1]$, then the *indirection* is $[0, 2, 5, 6, 1, 3, 4, 7]$, the *indexNew2Origin* array is: $[0, 2, 5, 6, 1, 3, 4, 7]$, while the *indexOrigin2New* array is $[0, 4, 1, 5, 6, 2, 3, 7]$.

4.2 MPI-based Library

The MPI-based library implements the programming interface in C++ with MPI. It employs both the domain decomposition and efficient graph partitioning algorithms to achieve optimal EP distribution and communication.

Storage of EPs and their Shadow Copies. Each process keeps different subsets of the EPs in the ensemble according to an EP distribution, which is determined by the root topology and an optimal EP distribution generated from METIS.

Ensemble Management. As the machine model is mapped on a distributed memory machine, the master thread is duplicated and resided across all the processes. Each process keeps an ensemble, which stores a subset of the EPs, their shadow copies, and references to topologies. The implementation of the *Ensemble* operations are described below:

1. *SC-Update*: It triggers communication among processes according to a topology specified in the operation and EP distribution algorithms.
2. *getNghbList*: It is a local operation implemented on each process. It only references local EPs stored in *loc_SC_Set* according to the topology specified in the operation.
3. *parallel*: Multiple processes executing EPs' member functions in parallel.
4. *collective*: A collective operation of EPs is translated into local collective operations and collective operations among MPI processes.

Topology Management. The topology is managed in a distributed fashion. Each process keeps the root topology, which maintains the neighbor EP list for the local EPs.

MultiBodyTopology. Each process keeps the root *MultiBodyTopology* topology, which maintains the neighbor EP list for all the local EPs. The generation of the neighbor EP list in the multi-body topology is based on the parallel Linked Cells algorithm, which is presented in Algorithm 2.

Algorithm 2. Creation of neighbor EP list

```
1. Forall ep in local process
    1) Get cell id localidCell of ep
    2) Get ids of neighbor cell localidCell
    3) Get neighbor cells local NghbCells
    4) Get EPs in the local nghbCells and determine whether the distance between the
EPs in local localidCell and ep is smaller than the cut-off radius
    5) If yes, put the address of EP
    6) Create neighbor EPs for ep
  End for
2. updateTopology() and go to Step 1
```

IrrGridTopology. Similar to the multi-body topology, each process keeps the root irregular grid topology, which maintains the neighbor EP list for the EPs in *loc_EP_Set* according to the id-based graph and EP distribution as well. The memory organization of local EPs and their SCs in a single process is shown in Fig. 3. The organization integrates the PARTI/CHAOS library [23]. The shadow copies of local EPs are allocated in the memory as an array, the SCs of remote EPs are stored after the local shadow copies with the indices from n to $n + m$.

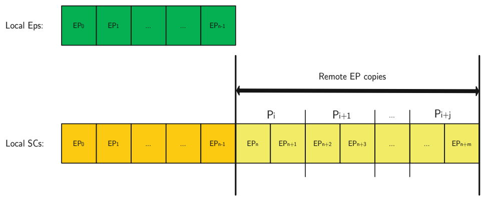


Fig. 3. Local EPs and SC organization

Communication Optimizations. Different communication optimizations are applied in the implementation of the MPI-based library.

1. *Aggregated send receive buffer management*: Each process keeps an aggregated send and receive buffer. The EPs in *loc_EP_Set* are copied into the aggregated send buffer, while EPs received from remote processes are stored in the aggregated receive buffer.
2. *Communication reduction*: It guarantees that an EP is only sent once while a group of remote EPs usually require the it for local computation. It can reduce the communication volume significantly.
3. *Communication coalescing*: A process collects many EPs destined for the same process into a single message, which is stored in the aggregated send buffer. The objective of communication coalescing is to reduce the number of message startups to avoid the “too many short messages” problem.
4. *Automatic adjustment of communication patterns according to the update of topologies*: For multi-body and irregular grid applications, the communication pattern is usually irregular and adaptive. The MPI-based library update the communication pattern accordingly based on runtime information.

5 Experimental Results

5.1 Overview

This section presents the experimental results of multi-body and irregular applications implemented by a manual program and an ensemble-based program. The manual program is implemented in C++, and parallelized with OpenMP and MPI, while the ensemble-based program is implemented by linking the libraries of the implementation framework.

5.2 Experiment Platform

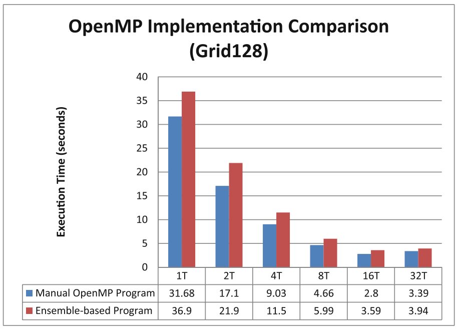
The experimental platform is a number of fat nodes on SuperMUC. A fat node is based on the Intel Westmere-EX processor. It is a shared memory NUMA machine with four sockets, each of which has one Intel Xeon Processor E7-4870 processor and 64 GB of memory. The processor has 10 cores running at the frequency 2.4 GHz with a peak performance of 9.6 GFlops.

5.3 Irregular Grid Applications

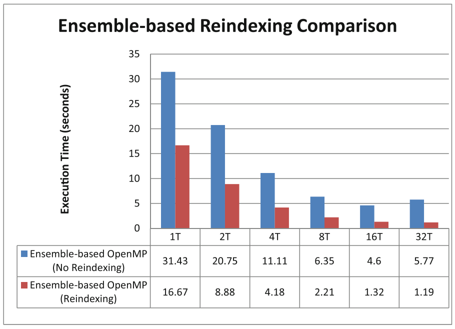
Overview. The computational kernel is a simplified version from FIRE [24]. The maximum number of the iterations N is set to 128. The grid size is $128 \times 128 \times 128$, each has 26 nearest neighbors based on the Moore neighborhood. The local values of a point at a time step are determined by the values of its neighbor points at its previous time step according to the arithmetical operations:

$$value^{N+1} = \frac{1}{numOfNeighbors+1}\left(\sum_{i=1}^{i=numOfNeighbors} Neighbor[i].value^{N} + value^{N}\right)$$

OpenMP Comparison. The execution time of the program on Grid128 using 2, 4, 8, 16, 32 threads is shown Fig. 4. Neither of the programs scale well when the number of threads is increased to 32. The main reason is that non-continuous memory accesses cause a large number of L2 and L3 cache misses. Therefore, we applies the re-indexing and data reallocation strategy to improve the performance on large irregular grids. We can see that the ensemble-based program implemented by the OpenMP-based library with the re-indexing and reallocation strategy scales well up to 32 threads and achieves much better performance than the ensemble-based implementation without re-indexing.



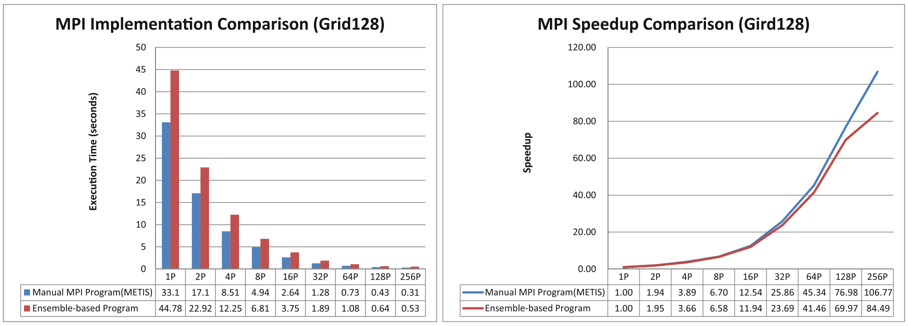
	1T	2T	4T	8T	16T	32T
🟦 Manual OpenMP Program	31.68	17.1	9.03	4.66	2.8	3.39
🟥 Ensemble-based Program	36.9	21.9	11.5	5.99	3.59	3.94



	1T	2T	4T	8T	16T	32T
🟦 Ensemble-based OpenMP (No Reindexing)	31.43	20.75	11.11	6.35	4.6	5.77
🟥 Ensemble-based OpenMP (Reindexing)	16.67	8.88	4.18	2.21	1.32	1.19

Fig. 4. Execution time of OpenMP programs (Color figure online)

MPI Comparison. The execution time and speedup curves comparison of both programs are shown in Fig. 5. It tells that the MPI-based library can get good performance while the number of processes increases with around 20 %



	1P	2P	4P	8P	16P	32P	64P	128P	256P
🟦 Manual MPI Program(METIS)	33.1	17.1	8.51	4.94	2.64	1.28	0.73	0.43	0.31
🟥 Ensemble-based Program	44.78	22.92	12.25	6.81	3.75	1.89	1.08	0.64	0.53

	1P	2P	4P	8P	16P	32P	64P	128P	256P
Manual MPI Program(METIS)	1.00	1.94	3.89	6.70	12.54	25.86	45.34	76.98	106.77
Ensemble-based Program	1.00	1.95	3.66	6.58	11.94	23.69	41.46	69.97	84.49

Fig. 5. Execution time comparison with MPI (Color figure online)

overhead compared to the manual MPI program. The ensemble-based program obtains comparative performance and its overhead is stable while scaling up to 256 processes. The execution time and speedup curves are shown in Fig. 5.

5.4 Molecular Dynamics Simulation

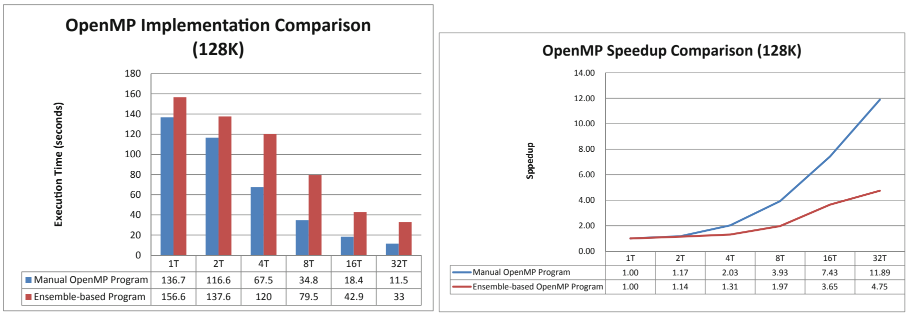
Overview. The computational kernel of the MD programs is based on the truncated Lennard-Jones(L-J) potential formula, and the simulation domain is a 3D cubic domain. Each molecule in the domain keeps a randomly generated position and interacts with its neighbor molecules located within the cut-off radius region. The positions of all the molecules are updated according to the molecule-to-molecule interactions and the equations of motion. In the experiments, the number of the molecules is set to 128K (131,072), the number of iteration steps is 8. The cut-off radius is set to 1, the size of the cubic simulation domain is 8.

OpenMP Comparison. The execution time and speedup curves of both programs is shown in Fig. 6. We can see that the overhead of the ensemble-based program becomes higher when the number of threads increases.

The overhead mainly originates from two aspects:

- The creation of the neighbor list is not efficient as expected because of its vector-based data structure.
- The parallel operation that doesn't scale very well because of memory bandwidth of the nodes on SuperMUC. Different from irregular grid applications, each molecule in an MD simulation typically has hundreds of neighbor molecules, which greatly increases the memory overhead.

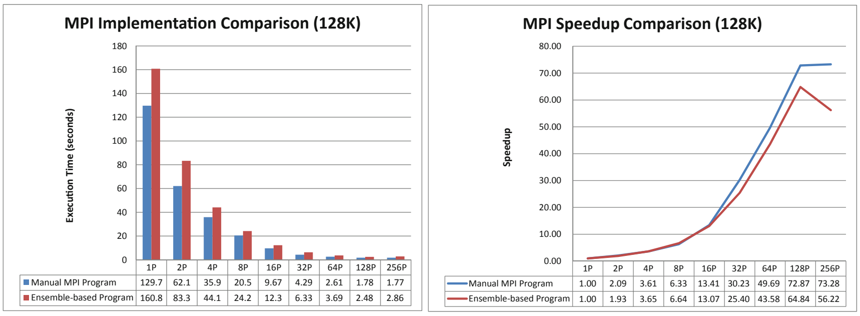
MPI Comparison. In order to balance the computational load, the manual program uses METIS to decompose the molecules according to the linked cells, while the ensemble-based program links to the MPI-based library. The execution time and speedup curves of both programs is shown in Fig. 7.



	1T	2T	4T	8T	16T	32T
🟦 Manual OpenMP Program	136.7	116.6	67.5	34.8	18.4	11.5
🟥 Ensemble-based Program	156.6	137.6	120	79.5	42.9	33

	1T	2T	4T	8T	16T	32T
Manual OpenMP Program	1.00	1.17	2.03	3.93	7.43	11.89
Ensemble-based OpenMP Program	1.00	1.14	1.31	1.97	3.65	4.75

Fig. 6. Execution time and speedup curves of OpenMP MD programs (Color figure online)



	1P	2P	4P	8P	16P	32P	64P	128P	256P
🟦 Manual MPI Program	129.7	62.1	35.9	20.5	9.67	4.29	2.61	1.78	1.77
🟥 Ensemble-based Program	160.8	83.3	44.1	24.2	12.3	6.33	3.69	2.48	2.86

	1P	2P	4P	8P	16P	32P	64P	128P	256P
Manual MPI Program	1.00	2.09	3.61	6.33	13.41	30.23	49.69	72.87	73.28
Ensemble-based Program	1.00	1.93	3.65	6.64	13.07	25.40	43.58	64.84	56.22

Fig. 7. Execution time and speedup curves of MPI MD programs (Color figure online)

The experimental results show that the ensemble-based implementations of multi-body and irregular applications are a bit slower than the manual implementations using C++ with OpenMP or MPI because of the internal function overheads. However, it improves the programming productivity in terms of the source code size, the coding method, and the implementation difficulty. For irregular grid applications, it saves around 80 % lines of code, while for multi-body applications, the percentage is more than 95 %.

6 Conclusion and Future Work

The fine granular programming scheme is applied to implement irregular scientific applications in a fine granular and SPMD fashion. The experimental results show that with acceptable and reasonable overhead, the ensemble-based programming improve the programming productivity and make parallel programming easier and more straightforward. In the future, we mainly focus on the support for more application areas, e.g., adaptive grid applications. In addition, the implementation for CPU+GPU hybrid architectures can also be exploited in the future in order to take advantages of hybrid programming.

References

1. Board, J.A., Hakura, Z., Elliott, W., Gray, D., Blanke, W., Leathrum, J.F.: Scalable implementations of multipole-accelerated algorithms for molecular dynamics. In: 1994 Proceedings of the Scalable High-Performance Computing Conference, pp. 87–94, May 1994
2. Boyd, D., Milosevich, S.: Supercomputing and drug discovery research. Perspect. Drug Discovery Des. **1**, 345–358 (1993). http://dx.doi.org/10.1007/BF02174534
3. Clementi, E., Chin, S., Corongiu, G., Detrich, J., Dupuis, M., Folsom, D., Lie, G., Logan, D., Sonnad, V.: Supercomputing and super computers: for science and engineering in general and for chemistry and biosciences in particular. In: Theophanides, T. (ed.) Spectroscopy of Inorganic Bioactivators. NATO ASI Series, vol. 280, pp. 1–112. Springer, Netherlands (1989)
4. Kremer, K.: Supercomputing in polymer research. In: Gentzsch, W., Harms, U. (eds.) HPCN-Europe 1994. LNCS, vol. 796, pp. 244–253. Springer, Heidelberg (1994)
5. Board, O.A.R.: OpenMP Application Program Interface. OpenMP, Specification (2011). http://www.openmp.org/mpdocuments/OpenMP3.1.pdf
6. Chapman, B., Jost, G., Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press, Cambridge (2007)
7. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI-The Complete Reference. The MPI Core, vol. 1, 2nd edn. MIT Press, Cambridge (1998)
8. Pacheco, P.S.: Parallel programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco (1996)
9. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge (1994)
10. NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture - Programming Guide (2007)
11. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008). http://doi.acm.org/10.1145/1365490.1365500
12. Schreiber, R.: An introduction to HPF. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 27–44. Springer, Heidelberg (1996)
13. Kennedy, K., Koelbel, C.: High performance fortran 2.0. In: Pande, S., Agrawal, D.P. (eds.) Compiler Optimizations for Scalable Parallel Systems. LNCS, vol. 1808, pp. 3–43. Springer, Heidelberg (2001)
14. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on C++. SIGPLAN Not. **28**(10), 91–108 (1993). http://doi.acm.org/10.1145/167962.165874
15. Kale, L.V., Ramkumar, B., Sinha, A.B., Gursoy, A.: The CHARM parallel programming language, system: Part I - Description of language features. Parallel Program. Lab. Tech. Rep. #95-02 **1**, 1–15 (1994)
16. Kale, L.V., Ramkumar, B., Sinha, A.B., Saletore, V.A.: The CHARM parallel programming language, system: Part II - The runtime system. Parallel Program. Lab. Tech. Rep. #95-03 **1**, 1–14 (1994)
17. Intel: TBB (Intel Threading Building Blocks). In: Padua, D. (ed.) Encyclopedia of Parallel Computing, p. 2029. Springer, Heidelberg (2011)

18. Russell, G., Keir, P., Donaldson, A.F., Dolinsky, U., Richards, A., Riley, C.: Programming heterogeneous multicore systems using threading building blocks. In: Guarracino, M.R., et al. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 117–125. Springer, Heidelberg (2011)
19. Molner, S.P.: The art of molecular dynamics simulation (Rapaport, D. C.). J. Chem. Educ. **76**(2), 171 (1999). http://pubs.acs.org/doi/abs/10.1021/ed076p171
20. Aarseth, S.J.: Gravitational N-Body Simulations. Cambridge University Press, Cambridge (2003). http://dx.doi.org/10.1017/CBO9780511535246
21. LRZ: SuperMuc petascale system (2012). https://www.lrz.de/services/compute/supermuc/systemdescription/
22. Karypis, G., Kumar, V., MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0 (1995). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.376
23. Das, R., shin Hwang, Y., Uysal, M., Saltz, J., Sussman, A.: Applying the CHPAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics, in Mississippi State University, Starkville, MS, pp. 45–56. IEEE Computer Society Press (1993)
24. Bericht, I., Gerndt, M.: Parallelization of the AVL FIRE benchmark with SVM-Fortran (1995)