The Journal of Open Source Software

TBFMM: A C++ generic and parallel fast multipole method library

Berenger Bramas[1, 2, 3]

1 CAMUS Team, Inria Nancy **2** Strasbourg University **3** ICPS Team, ICube

DOI: 10.21105/joss.02444

Software

- Review
- Repository
- Archive

Editor: Jack Poulson

Reviewers:

- @pitsianis
- @Himscipy
- @sarats

Submitted: 20 May 2020
Published: 20 July 2020

Summary

TBFMM, for task-based FMM, is a high-performance package that implements the parallel fast multipole method (FMM) in modern C++17. It implements parallel strategies for multicore architectures, i.e. to run on a single computing node. TBFMM was designed to be easily customized thanks to C++ templates and fine control of the C++ classes' inter-dependencies. Users can implement new FMM kernels, new types of interacting elements or even new parallelization strategies. As such, it can be used as a simulation toolbox for scientists in physics or applied mathematics. It enables users to perform simulations while delegating the data structure, the algorithm and the parallelization to the library. Besides, TBFMM can also provide an interesting use case for the HPC research community regarding parallelization, optimization and scheduling of applications handling irregular data structures.

Background

The fast multipole method (Greengard & Rokhlin, 1987) has been classified as one of the most important algorithms of the 20th century (Cipra, 2000). The FMM algorithm was designed to compute pair-wise interactions between N particles, which belong to the class of n-body problems. It reduces the complexity from a quadratic (N elements interact with N elements) to a quasi-linear complexity. The central idea of the FMM is to avoid computing the interactions between all the elements by approximating the interactions between elements that are far enough. To make this possible, the algorithm requires the potential of the interactions to decrease as the distance between interacting elements increases. In addition, the algorithm also requires that the kernel to approximate far interaction exists as providing an approximation kernel for a physical equation can be challenging. Internally, the FMM is usually implemented with a tree that is mapped over the simulation box. A cell, i.e. a node of the tree, represents a part of the simulation box and is used by the algorithm to factorize the interactions between elements. The FMM was later extended for different types of physical simulations and different approximation kernels (Barba & Yokota, 2011; Blanchard et al., 2015a; Blanchard, Coulaud, Etcheverry, Dupuy, & Darve, 2016; Darve & Havé, 2004; Darve, Messner, Schanz, & Coulaud, 2013; Frangi, Faure-Ragani, & Ghezzi, 2003; Malhotra & Biros, 2015; Pham, Mouhoubi, Bonnet, & Chazallon, 2012; R. Sabariego, 2004; R. V. Sabariego et al., 2004).

The FMM algorithm is based on six operators with names that respect the format X2Y, where X represents the source of the operator and Y the destination. The operators are P2M, M2M, M2L, L2L, L2P and P2P, where P means particle, M multipole and L local. The term particle is used for a legacy reason, but it represents the basic interaction elements that interact and for which we want to approximate the interactions. The multipole part represents the aggregation of potential, i.e. it represents what is emitted by a sub-part of the simulation box, whereas

the local part represents the outside that is emitted onto a sub-part of the simulation box. The different operators are schematized in Figure 1.

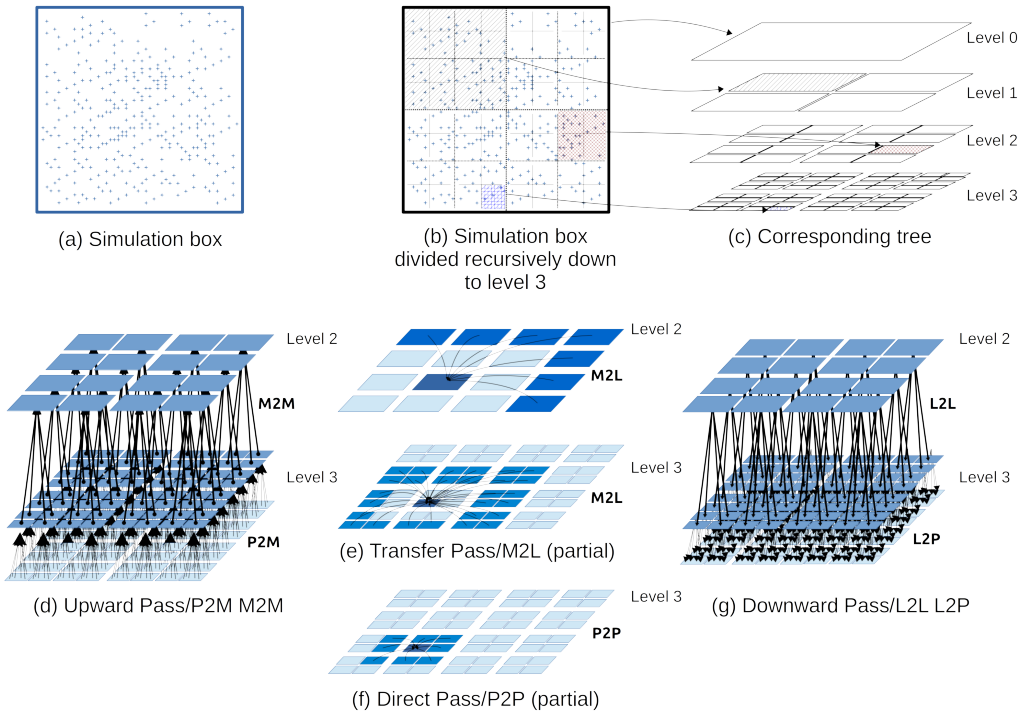


Figure 1: Illustration of the FMM algorithm. (a,b,c) The building of the octree. (d,e,f,g) The FMM algorithm and its operators.

Because the FMM is a fundamental building block for many types of simulation, its parallelization has already been investigated. Some strategies for parallelizing over multiple distributed memory nodes have been developed using classical HPC technologies like `MPI` (Forum, 1994) and fork-join threaded libraries (Bramas, 2016). However, when using a single node, it has been demonstrated that fork-join schemes are less efficient than task-based parallelization on multicore CPUs (Agullo et al., 2014). This is because some stages of the FMM have a small degree of parallelism (for instance at the top of the tree), while others have a high degree of parallelism. For instance, the `P2P` in the direct pass has a significant workload available from the early beginning of each iteration. The task-based method can interleave the different operators, hence to balance the workload across the processing units and to spread the critical parts over time. Moreover, the task-based method is well designed for handling heterogeneous architecture (Agullo et al., 2016) and has demonstrated a promising performance on distributed memory platforms too (Agullo, Bramas, Coulaud, Khannouz, & Stanisic, 2017).

In a previous project called `ScalFMM`, we have provided a new hierarchical data structure called group-tree (or block-tree), which is an octree designed for the task-based method (Bramas, 2016). The two main ideas behind this container are (1) to allocate and manage several cells of the same level together to control the granularity of the tasks, and (2) to store the symbolic data, the multipole data, and the local data in a different memory blocks. This allows us to move each block anywhere on the memory nodes and to declare the dependencies on each sub-part.

A schematic view of the group-tree is given in Figure 2.

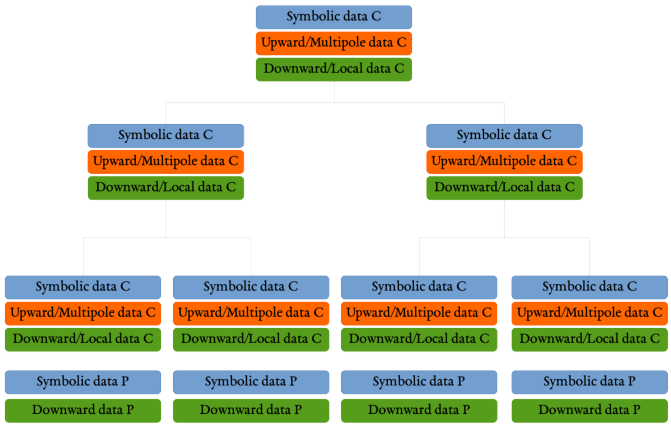


Figure 2: Group-tree schematic view. Several cells/leaves are managed together inside a block. Also, the symbolic, multipole and local data are allocated separately to allow declaring the data accesses on each sub-part.

Statement of need

The FMM is a major algorithm but it remains rare to have it included in HPC benchmarks when studying runtime systems, schedulers or optimizers. The principal reason is that it is tedious to implement and requires a significant programming effort when using the task-based method together with the group-tree. However, it is an interesting, if not unique, algorithm to study irregular/hierarchical scientific method. For the same reason, it is difficult for researchers in physics or applied mathematics to implement a complete FMM library and to optimize it for modern hardware, especially if their aim is to focus on approximation kernels. Therefore, `TBFMM` can be useful for both communities.

Among the few existing FMM libraries, `ScalFMM` is the closer package to `TBFMM`. `ScalFMM` supports lots of different parallel strategies, including fork-join implementations, and it contains several experimental methods. Consequently, `ScalFMM` has around 170K lines of code, for only 50K for `TBFMM`. Moreover, it needs several external dependencies and does not benefit from the new standard `C++` features that could improve code readability. Besides, it only works for 3D problems, whereas `TBFMM` can work for an arbitrary dimension. These have been the main motivations to re-implement a lightweight FMM library from scratch that only supports task-based parallelization.

However, the interface of the kernels is very similar in both libraries, such that creating a kernel for `ScalFMM` or `TBFMM` and porting it to the other library is straightforward.

Features

Genericity

`TBFMM` has a generic design thanks to the heavy use of `C++` templates. The tree and the kernel classes are independent of each other and from the algorithm. The algorithm class has to be templatized in order to know the type of the kernel, and its core `execute` method has to be templatized with the type of the tree. The algorithm takes the elements from the tree and passes it to the kernel, such that a kernel itself never accesses the tree. This is illustrated by Figure 3.

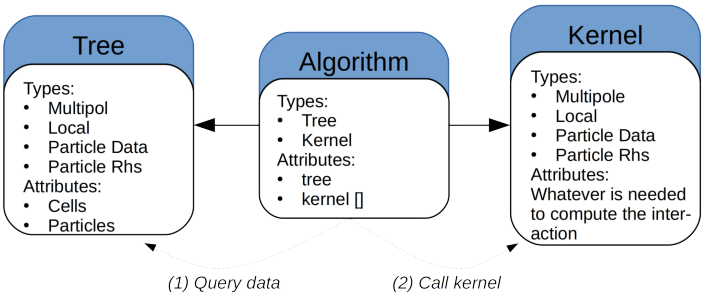


Figure 3: `TBFMM` design overview. The `Types` of each class should be templatized, at the exception of the types of the kernel where it is optional. The algorithm has to be selected among different variants (sequential, parallel OpenMP or parallel SPETABARU).

Tree

`TBFMM` uses the group-tree where several cells of the same level are managed together. Users can select the size of the groups, which impacts the size of the tasks, however, `TBFMM` also provides a simple heuristic to automatically find a size, which should provide efficient executions. Also, the tree class provides different methods to iterate on the cells/leaves as any container, such that it is possible to work on the elements of the tree with an abstraction mechanism and without knowing how it is implemented internally.

Kernel

As stated in the objectives, `TBFMM` is a tool for scientists from physics and applied mathematics to create new kernels. `TBFMM` offers a convenient way to customize the kernel and to benefit from the underlying parallelization engine automatically. With this aim, a user has to create a new kernel that respects an interface, as described by the package documentation. The current package contains two FMM kernels, the `rotation` kernel based on the rotation-based operators and the spherical harmonics (Dachsel, 2006; Haigh, 2011; White & Head-Gordon, 1994, 1996), and the `uniform` kernel based on Lagrange interpolation (Blanchard et al., 2015a, 2015b, 2016).

Parallelization

`TBFMM` has two task-based parallel algorithms based on two runtime systems: `OpenMP` version 4 (Board, 2013) and `SPETABARU` (Bramas, 2019). Both are optional, such that the library can be compiled even if the compiler does not support `OpenMP` or if the `Git` sub-module for `SPETABARU` has not been activated. `OpenMP` is an API that evolves slowly, which maintains backward compatibility and which is implemented by different libraries that respect the standard. On the other hand, `SPETABARU` is our task-based runtime system that we use for research, and which continuously evolves. The data accesses of the FMM operators in `write` are usually commutative (Agullo, Aumage, Bramas, Coulaud, & Pitoiset, 2017). While `SPETABARU` supports commutative `write` access, `OpenMP` only supports it from version 5 with the `mut exinout` data access. `OpenMP` version 5 is currently not fully supported by the compilers, however, when a compiler that supports this access will be used with `TBFMM`, the `mutexinout` will be activated automatically.

Periodicity

The periodicity consists of considering that the simulation box is repeated in all directions, as shown by Figure 4. Computing the FMM algorithm with periodicity is usually done in two steps. In the first one, the regular algorithm and tree are used, however, when the algorithm needs cells outside of the boundaries, it selects cells at the opposite side of the simulation box. While in the second step, a numerical model is used to compute a potential that represents the world outside the simulation box. Such a model could be the Ewald summation (Rokhlin & Wandzura, 1994).

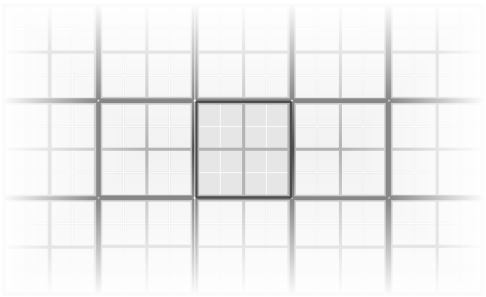
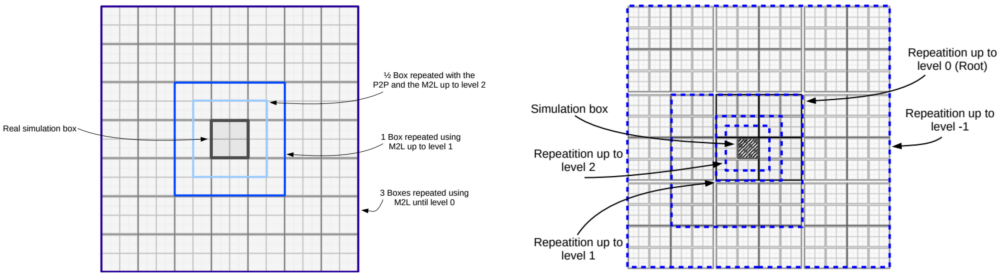


Figure 4: In the periodic FMM, the simulation box is considered to be in the middle of an infinite volume of the same kind.

In `TBFMM`, we have implemented a different approach, which is a pure algorithmic strategy (Bramas, 2016). The idea is to consider that the real simulation box is a sub-part of a larger simulation box, i.e. that the real tree is a branch of a larger tree. Then, instead of stopping the FMM algorithm at level 2, we continue up until the root where the multipole part of the root represents the complete simulation box. We use it by continuing the FMM algorithm partially above the root, by aggregating the cells together multiple times. By doing so, we have several advantages. This method needs nothing more than a FMM kernel, which is expected to be the same as the one used without periodicity. Therefore, the method is generic and can work with any FMM kernel. Moreover, the accuracy of the method relies fully on the FMM kernel. Figure 5 shows how the simulation box is repeated with this method.



(a) Repetitions with periodic FMM up to level 0

(b) Repetitions with periodic FMM up to level -1

Figure 5: How the simulation box is repeated when using a periodic FMM algorithm.

Vectorization (Inastemp)

When implementing a kernel, some parts can rely on well-optimized numerical libraries, such as BLAS or FFTW, however, others might be implemented directly in `C/C++`. In this case, it usually provides a significant improvement in performance to vectorize the code, which allows benefiting from the SIMD capability of modern CPUs. With this aim, `TBFMM` can include

a vectorization library called `Inastemp` (Bramas, 2017) by simply cloning the corresponding `Git` sub-module. Using `Inastemp`, it is possible to write a single code with an abstract vector data type and to select at compile time the desired instruction set depending on the CPU (`SSE`, `AVX`, `SVE`, etc.). In the current version of `TBFMM`, the `P2P` operator of the two kernels that are provided for demonstration is vectorized with `Inastemp`.

Performance

In Figure 6, we provide the parallel efficiency of `TBFMM` for a set of particles that are randomly distributed in a square simulation box. The given results have been computed using the `uniform` kernel and the two runtime systems OpenMP (GNU libomp) and SPETABARU. With OpenMP, there is no reduction in the execution time with more than 8 threads, which appears as a significant drop in the parallel efficiency. This happens because we use OpenMP 4.5 that does not support `mutexinout`, the commutative data access, hence the resulting degree of parallelism is too limited to feed all the cores available.

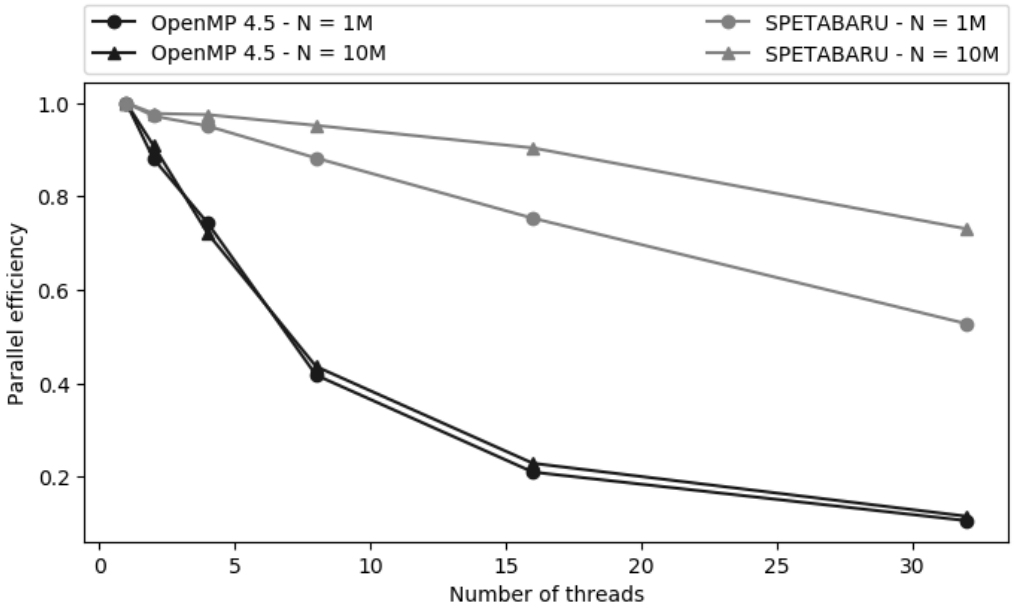


Figure 6: Parallel efficiency for `TBFMM` using the OpenMP and SPETABARU runtime systems, and the uniform kernel (order $= 8$). Test cases: two simulations of one and ten millions of particles randomly distributed in a cube. Hardware: $2 \times$ Intel Xeon Gold 6240 CPU at 2.60GHz with 16 cores each and cache of sizes L1/32K, L2/1024K, L3/25344K.

Conclusion & Perspective

`TBFMM` is a lightweight FMM library that could be used for research in HPC and applied mathematics. We will include it in our benchmarks to evaluate scheduling strategies, but also to validate new approaches to develop numerical applications on heterogeneous computing nodes. Indeed, we would like to offer an elegant way for users to add GPU kernels while delegating most of the complexity to `TBFMM` and `SPETABARU`. We also plan to provide an `MPI` version to support distributed memory parallelization shortly.

Acknowledgements

Acknowledgment: Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine.

References

Agullo, E., Aumage, O., Bramas, B., Coulaud, O., & Pitoiset, S. (2017). Bridging the gap between openmp and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, *28*(10), 2794–2807. doi:10.1109/tpds.2017.2697857

Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., & Takahashi, T. (2014). Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, *36*(1), C66–C93. doi:10.1137/130915662

Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., & Takahashi, T. (2016). Task-based fmm for heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, *28*(9), 2608–2629. doi:10.1002/cpe.3723

Agullo, E., Bramas, B., Coulaud, O., Khannouz, M., & Stanisic, L. (2017). *Task-based fast multipole method for clusters of multicore processors* (Research Report No. RR-8970) (p. 15). Inria Bordeaux Sud-Ouest. Retrieved from https://hal.inria.fr/hal-01387482

Barba, L., & Yokota, R. (2011). ExaFMM: An open source library for fast multipole methods aimed towards exascale systems. *Boston: Boston University. Retrieved from barbagroup: http://barbagroup. bu. edu*.

Blanchard, P., Coulaud, O., & Darve, E. (2015a). Fast hierarchical algorithms for generating gaussian random fields.

Blanchard, P., Coulaud, O., Darve, E., & Bramas, B. (2015b). Hierarchical randomized low-rank approximations. In.

Blanchard, P., Coulaud, O., Etcheverry, A., Dupuy, L., & Darve, E. (2016). An efficient interpolation based fmm for dislocation dynamics simulations. In.

Board, O. A. R. (2013, July). OpenMP application program interface. Retrieved from https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

Bramas, B. (2016). *Optimization and parallelization of the boundary element method for the wave equation in time domain* (PhD thesis). Bordeaux.

Bramas, B. (2017). Inastemp: A novel intrinsics-as-template library for portable simd-vectorization. *Scientific Programming*, *2017*. doi:10.1155/2017/5482468

Bramas, B. (2019). Increasing the degree of parallelism using speculative execution in task-based runtime systems. *PeerJ Computer Science*, *5*, e183. doi:10.7717/peerj-cs.183

Cipra, B. A. (2000). The best of the 20th century: Editors name top 10 algorithms. *SIAM news*, *33*(4), 1–2.

Dachsel, H. (2006). Fast and accurate determination of the wigner rotation matrices in the fast multipole method. *The Journal of Chemical Physics*, *124*(14), 144115. doi:10.1063/1.2194548

Darve, E., & Havé, P. (2004). A fast multipole method for maxwell equations stable at all frequencies. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, *362*(1816), 603–628.

Darve, E., Messner, M., Schanz, M., & Coulaud, O. (2013). Optimizing the black-box fmm for smooth and oscillatory kernels. In.

Forum, M. P. (1994). *MPI: A message-passing interface standard*. USA: University of Tennessee.

Frangi, A., Faure-Ragani, P., & Ghezzi, L. (2003). Coupled fast multipole method-finite element method for the analysis of magneto-mechanical problems. In *Proceedings of the sixth french national congress" calcul des structures* (pp. 273–280).

Greengard, L., & Rokhlin, V. (1987). A fast algorithm for particle simulations. *Journal of Computational Physics*, *73*(2), 325–348. doi:10.1016/0021-9991(87)90140-9

Haigh, A. (2011). Implementation of rotation-based operators for fast multipole method in x10. Austrailian National University.

Malhotra, D., & Biros, G. (2015). PVFMM: A parallel kernel independent fmm for particle and volume potentials. *Communications in Computational Physics*, *18*(3), 808–830. doi:10.4208/cicp.020215.150515sw

Pham, A. D., Mouhoubi, S., Bonnet, M., & Chazallon, C. (2012). Fast multipole method applied to symmetric galerkin boundary element method for 3D elasticity and fracture problems. *Engineering analysis with boundary elements*, *36*(12), 1838–1847. doi:10.1016/j.enganabound.2012.07.004

Rokhlin, V., & Wandzura, S. (1994). The fast multipole method for periodic structures. In *Proceedings of ieee antennas and propagation society international symposium and ursi national radio science meeting* (Vol. 1, pp. 424–426 vol.1). doi:10.1109/aps.1994.407723

Sabariego, R. (2004). The fast multipole method for electromagnetic field computation in numerical and physical hybrid systems.

Sabariego, R. V., Gyselinck, J., Geuzaine, C., Dular, P., & Legros, W. (2004). Application of the fast multipole method to hybrid finite element–boundary element models. *Journal of Computational and Applied Mathematics*, *168*(1), 403–412. doi:10.1016/j.cam.2003.12.011

White, C. A., & Head-Gordon, M. (1994). Derivation and efficient implementation of the fast multipole method. *The Journal of Chemical Physics*, *101*(8), 6593–6605. doi:10.1063/1.468354

White, C. A., & Head-Gordon, M. (1996). Rotating around the quartic angular momentum barrier in fast multipole method calculations. *The Journal of Chemical Physics*, *105*(12), 5061–5067. doi:10.1063/1.472369