A Compiler-based Technique for Simple, Efficient Message Packing

Anonymous Authors
Institution Withheld
Email Withheld

Abstract

The communication of data is fundamental to parallel programming, and in any programming model that supports user-defined types and multiple address spaces, these types must be marshallable and unmarshallable in order for them to be well-integrated into the model. However, the design and implementation of data marshalling and unmarshalling in many common frameworks leads to an undue burden on the programmer, requiring time and effort to accomplish tasks that can be done more accurately and efficiently by a compiler. We describe the use of compiler techniques to produce efficient marshalling code without programmer intervention. We employ these techniques in the context of a parallel programming framework which is primarily implemented as a user-level library, and describe the benefits and costs of such an approach.

1. Introduction

High Performance Computing (HPC) is a notoriously challenging software engineering challenge. The complexities involved in writing fast, scalable applications are enormous and growing every year. At the same time, many of the authors of HPC software are experts in their particular application domain rather than experts in programming, much less parallel programming.

Considering the intrinsic difficulties of HPC and the demands upon HPC programmers, it can be no surprise that programmer productivity in this area is notoriously poor [14, 16, 20, 21]. Sadly, no dramatic solution to this problem has been found, and none seems likely to present itself in the near future.

Under these circumstances, we must strive to relieve the programmer of as many burdens as is practically possible. Although the intrinsic complexities of HPC software may always remain, we can at least aim to remove as much of the tiresome drudgery of programming as we can, eliminating boilerplate code wherever possible.

One possible target for this effort is the passing of program data between address spaces in distributed memory applications. This process requires the programmer to convert application data into a communication-ready format, typically a packed buffer, on the sending side (this process is known as *marshalling*, *serialization*, or *packing*), then reconstitute the data on the receiving side (*unmarshalling*, *deserialization*, or *unpacking*).

Empirical studies have indicated that shared memory programming is more productive than distributed memory [11], and one of the factors that weighs against distributed memory programming is the need to pack and unpack application data. Any disagreement between packing code and the corresponding unpacking code can lead to subtle bugs, and the code must be carefully maintained whenever the data being transmitted changes.

In object-oriented programs, the data being transmitted will typically include user-defined types. In most programming models with explicit messaging, the programmer must provide code to handle the packing and unpacking of these types. This support for managing the communication of user-defined types is notable for requiring the programmer to manually specify information that the compiler itself must already know–that is, the types of the variables involved and how they are laid out in memory.

In this paper we present our approach to the problem of packing and unpacking application data in a distributed memory environment. We avoid the need for the programmer to manually specify how data structures will be packed and unpacked. We even avoid the need for the programmer to specify which fields of a user-defined type should be packed and which do not need to be sent and can be safely excluded. We do this while producing efficient packing and unpacking code which does not require maintenance when application datatypes or communication patterns are changed. We do this by leveraging simple compiler analysis and code generation. In the remainder of this paper we give our rationale

for implementing this system, describe the system itself and its advantages and disadvantages, and give two case studies. We then list related and future work and conclude.

2. Approach

There are many methods by which the code which transmits application data can be created. Perhaps the simplest approach is for the programmer to do the work manually. This largely consists of determining the size of the data to be sent, allocating a buffer of the appropriate size, and then copying the relevant application data into the buffer.

The advantage of this technique is that it is completely customizeable. If a subfield of some user-defined type is needed by a receiver in some portions of an application but not others, the programmer can account for this fact directly. If several variables are known to be contiguous in memory, they can be copied as a block rather than individually.

However, the drawbacks of this approach are obvious. It is a lot of repetitive work to specify all the data that an application transmits in detail, and whenever application data structures change, all the packing and unpacking code has to change with it. It is also error prone, and there is no easy way of verifying that the packing and unpacking is bug-free. While this approach may be feasible, and even high-performance given time and effort, it is extremely poor for productivity.

Alternatively, the programmer may use a library to assist with creating the code. This approach has the advantage that well-designed libraries can significantly ease the process of writing packing and unpacking code while increasing confidence in that code's correctness. These libraries range from the relatively spartan to full-featured libraries such as Boost.Serialization which include features for cyclic data structures and conditional packing.

However, these libraries typically lack the flexibility to efficiently change the way that an object is packed based on application context. Each field of a type must be either always included or always excluded, leading to inefficiencies. They also require at least some level of intervention by the programmer to integrate their data structures with the library in questin.

A more automatic approach is to use the information available at compile time to generate packing code that guarantees type safety while eliminating the need for manual intervention by the programmer. Because the compiler knows the data layout of each type it can effectively generate packing and unpacking code that does not require updates from the programmer. However, a straightforward implementation will still pack data that may not be needed on the receiving side. The programmer can specify which fields to skip, but this requires user intervention and doesn't allow for the possiblity that some fields may be needed in one situation but not in another.

One goal of our work is to produce a tool that can be genuinely useful to application developers. However, this goal is somewhat in tension with our decision to do this work within a compiler and language framework built on a programming model that is not MPI. Since the great majority of HPC applications run on MPI, it would be a natural target for practical productivity work.

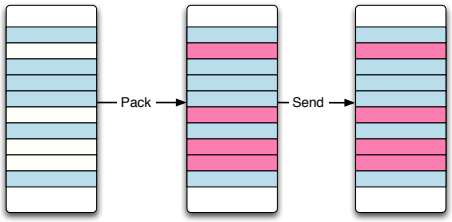
While these techniques could be applied in the context of MPI, we have several reasons for our choice of platform. First, the object-oriented nature of the Charm programming model is a natural fit for our method. Parallel objects abound in any Charm application, and the packing and unpacking of these objects is therefore a more prominent issue than it might be in an equivalent MPI program. Second, the existence of a compiler infrastructure built on Charm made the implementation of our ideas straightforward. Simply parsing arbitrary C++ is a complex task, and avoiding this large initial effort is a large benefit. Third, while Charm is not nearly as successful or widespread as MPI, there are significant HPC applications using Charm in production [2, 12, 24], so there is still scope for practical impact using our implementation.

3. Implementation

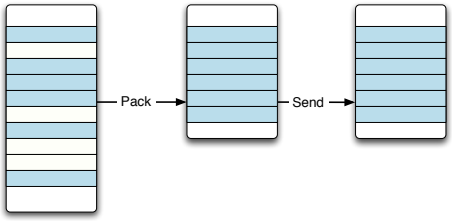
One of the benefits of our approach is that it does not require any complex or time-intensive compiler analysis. For each remotely invokable method in our application, we wish to produce a function that will pack its arguments, discarding any data which can be proven to be unused. The primary question to be answered is, which variables can be discarded?

Fortunately, there is a simple compiler analysis that answers this question. Since the function does not interact with its unused fields, the values in those fields are not used in any control flow path that begins at the head of the functions control flow graph. Thus, the function argument fields that are not needed in the body of the function are simply those fields that are not live at the start of the function. Live variable analysis is a well-known and well-studied algorithm [15], so implementation is straightforward. We perform interprocedural analysis where possible, and when code from external libraries is invoked we pessimistically assume that all fields of all arguments to external functions are used.

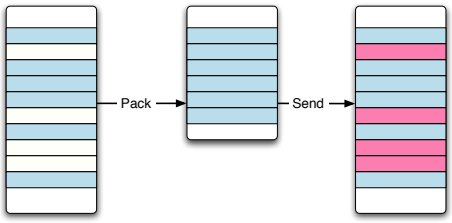
We treat each user-defined type as a set of elements, with each element corresponding to one field. The output of the live variable analysis is the set of all elements which are live at the function's beginning. Using this set we generate packing code specific to this function which copies each live variable into a buffer, and corresponding unpacking code which reconstitutes the function arguments on the receiving side. To minimize the complexity of our implementation we recreate the full types of all function arguments. This is potentially wasteful of memory, as shown in figure 1(c). A better approach would be to transform the receiving function



(a) The simplest approach is to simply pack the entire data structure regardless of which fields are needed and which are not. This is wasteful of space but maintains encapsulation.



(b) By writing a custom packing routine, the programmer can ensure that no data is unnecessarily transmitted at the cost of breaking encapsulation at the receiving side.



(c) Our technique packs only required fields, but reconstitutes this data on the receiving side as though it was the full object. This maintains encapsulation without wasting bandwidth, but does incur memory overhead on the receiving side.

Figure 1. Three approaches to message packing and unpacking. The leftmost box represents a data structure to be sent, and the rectangles inside it represent its fields. The middle box represents the message buffer, and the rightmost box represents the unpacked data at its destination. Fields that are required by the receiving side are colored blue, while wasted memory is colored pink.

so that instead of expecting the set of arguments specified by the programmer, it instead expects the set of variables that it actually uses. We do not believe that this transformation is difficult in theory, and have left it for future work.

In the base programming model that we are extending, each type has an associated packing function, and *FIXME*

4. Case Studies

To get a clear idea of how this all works in practice, it is helpful to look at message packing in the context of actual applications. One of the principal advantages of our technique is that it allows the programmer to describe communication in terms natural, high-level objects with semantic meaning rather than simply enumerating the data that will be consumed by the receiving function. However, this benefit cannot be demonstrated on tiny programs like microbenchmarks, because by their nature they are stripped down to the bare essentials needed to perform one task effectively. Thus there are typically no high-level objects that are used in multiple different ways in different contexts, as one would expect in a more realistic application.

To show how our message packing scheme works in an application context without introducing the full complexities and size of a real production HPC code, we present two case studies taken from the examples provided with the Charm runtime system. These are scaled-down, simplified applications that maintain the structure of more sophisticated scientific codes, but in a smaller and simpler package.

4.1 Molecular Dynamics

Charm is best known for NAMD [2], a popular molecular dynamics program in common use at national supercomputing sites. However, NAMD is large and complex, and we do not have the resources that would be required to port NAMD to Charj. However, Charm provides an example molecular dynamics program, named Molecular2D, with similar overall structure to NAMD but with greatly simplified two-dimensional physics. Since this program is provided for pedagogical purposes we might expect it to be written in a way that maximizes clarity at the cost of performance, and in fact this is the case, at least when it comes to message packing.

The primary data structures used in Molecular2D are Particles, which represent the physical objects being modeled, and Patches, which represent a region of space which may contain any number of particles. Listing 1 shows the full definition of the Particle type, which mostly consists of information regarding the physical properties of the particle.

The application simulates the motion of these particles over a series of timesteps. In each step, particles within a certain radius exert forces on one another, affecting the position, velocity and acceleration of each. Objects called computes are responsible for managing the interactions between neighboring patches. Each patch sends data regarding its particles to compute objects so that they can determine the ef-

Listing 1. The central particle data structure used by Molecular2D, and its accompanying PUP method.

```
class Particle{
  public:
    int id;
    double mass;      // mass
    double pos[2];    // position
    double f[2];      // force
    double a[2];      // acceleration
    double v[2];      // velocity

    void pup(PUP::er &p) {
      p | id;
      p | mass;
      p(pos, 2);
      p(f, 2);
      p(a, 2);
      p(v, 2);
    }
};
```

Listing 2. Methods in Molecular2D which receive Particle objects from remote senders. Each takes a list of particles from a remote object which has packed the particle data into a buffer and delivered it to the current patch.

```
class Patch {
    void updateForces(
            vector<Particle> particles);
    void updateParticles(
            vector<Particle> updates);
    // ...
};
```

Listing 3. A pup function equivalent to the packing code Charj generates for the `updateForces` method.

```
void Particle::pup(PUP::er &p) {
  p(f, 2);
}
```

fect of those particles on particles belonging to other nearby patches. As the position of a particle changes, it may be migrated from one patch to another.

Listing 2 shows the signatures of the functions used by each patch to communicate particle information during each timestep. These are both remotely invoked functions, so their arguments have been marshalled by potentially remote elements. The `updateForces` function is called by a compute which has calculated force contributions to local particles. The function's argument is a list of particles corresponding to local particles which have forces exerted on them by particles from another patch. The function simply updates the net force on its own particles based on the information it receives from the compute object. The `updateParticles` function migrates particles which have moved outside a patch boundary to the appropriate neighboring patch. This function's argument is a list of formerly remote particles which have moved within the boundaries of the patch during the last timestep.

Semantically, both of these functions operate on a combination of local and remote particle data, so it is natural that they each receive a list of particles as their argument. However, their use of the particle data they receive is quite different. In the case of `updateParticles`, the particles in the list are migrating to a new patch, and so none of their data can be omitted–each particle will need all of its fields in the next timestep in its new patch. However, this is not the case for `updateForces`. These particles are not migrating, only contributing to the forces exerted on some local particles. Indeed, if we look at the function body in detail, we can see that the only fields of the received particles that used are the forces. The force members represent 16 bytes out of a total of 76 bytes per particle, so nearly 80% of the data transmitted to `updateForces` is pure waste.

In translating this code to Charj, the functions remain mostly unchanged, except that the pup function is now unnecessary. However, the actual communication that takes place is much different. During compilation, `updateParticles` and `updateForces` are each analyzed to determine which fields of their arguments are potentially used. In the case of `updateForces` the forces are the only particle components that can possibly be read, so method-specific packing code equivalent to listing 3 is generated. In the case of `updateParticles`, the elements of the argument array are added to a data structure belonging to the patch, and from that point on any of their fields could be accessed by Patch methods. Therefore the packing code generated by Charj for this function is equivalent to the full pup method of the original application.

4.2 N-Body Simulation

The second application we consider is a modified version of the Barnes-Hut N-body algorithm [1] from the well-known SPLASH-2 suite [19]. The modifications are limited to porting the application to use the Charm runtime. The kernel and overall structure of the application remain unchanged.

In this application, a volume of space containing particles is divided into regions using an oct-tree, with each leaf of the tree representing a volume of space that contains an approximately the same number of particles, though the size of these volumes may vary greatly depending on the spatial particle distribution. Then when performing n-body

Listing 4. A method in the Barnes-Hut application that passes information down the tree. It receives several arguments, each of which is a field of the parent object.

```
void recvRootFromParent(uint8_t root_id,
    double rx, double ry,
    double rz, double rs);
```

Listing 5. A Charj method signature corresponding the the method in listing 4.

```
void recvRootFromParent(TreePiece parent);
```

calculations, only particles from nearby volumes must be considered individually, with the contribution of particles from remote volumes only approximated.

The primary communication that takes place in this application is the passing of interaction data up and down the tree. The tree is decomposed into disjoint segments called TreePieces, and data is communicated between pieces via remote invocation of a few methods. Actual transfer of particle data simply uses a vector of particle information in much the same way as the molecular dynamics application described previously. However, information about parent-child relationships within the tree is communicated using other methods of the TreePiece object, such as `recvRootFromParent`.

As shown in listing 4, `recvRootFromParent` takes several arguments describing its parent. What is not obvious from the method signature, however, is that each of the arguments comes from a field of the same parent object. However, it is completely impractical to send the entire parent object, because this object contains dozens of fields and a huge amount of data that should not be transmitted.

While the solution adopted by the application of simply splitting out the required data and sending it separately is vastly more efficient, it obscures the origin of the data and the relationship between its arguments. One could preserve this information to some extent by creating a custom type that encapsulates just the information needed for this function, but that approach has high overhead for the programmer, especially in large applications or when an application is being refactored and its arguments change.

Listing 5 shows a Charj method signature for the same function. Within the method, uses of `rx` are replaced by `parent.rx`, `ry` by `parent.ry` and so on. This simplifies the method signature, making it easier to see how the function works at a glance. In this case the improvement isn't life-changing, but in a larger and more complicated application methods may have dozens of parameters, some subset of which come from a common object and others of which do not. In those cases the simplification may represent a dramatic easing of the burden on the programmer.

5. Related Work

A large amount of work has been done on data marshalling, both on improving efficiency and on reducing the burden on the programmer. Systems such as Sun RPC [22] provided for marshalling of C structs, using a high-level specification for communcication in concert with a stub compiler. Later systems such as CORBA [5] extended this functionality into the object-oriented world. Later work improved the efficiency of generated marshalling code by dynamically choosing between runtime interpretation of data descriptions and compilation [6, 8, 17]. However, these systems all require the programmer to explicitly describe the data to be marshalled and do not attempt to determine if any unused data is being transmitted.

More recently there have been several approaches published for providing serialization of C and C++ data structures in MPI applications. C++2MPI [10] and the MPI Preprocessor [18] are both capable of automatically extracting `MPI_Datatype` definitions from C and C++ types. They generate a list of offsets describing the location of all data to be marshalled relative to the base address of the user's data. However, they are limited to marshalling the structure in its entirety and do not handle the case of omitting unneeded data, even in simple cases where the unneeded data does not depend on application context.

AutoMap and AutoLink [9] are also tools that extract MPI datatypes from user code. However, they are limited to C and require the programmer to annotate which fields to pack and which to omit.

Software engineering tools focused on boosting productivity through refactoring have also targeted data marshalling as an area where productivity gains can be had [7]. In [23], Tansey and Telvich describe a graphical tool for generating marshalling code in an MPI context. They allow for multiple versions of the marshalling code to account for the case where different data is needed by the reciever in different application contexts, much as we do here. However, they rely on the user to manually specify which fields will be packed and which will be omitted in each case, whereas we generate all marshalling code automatically and use compiler analysis to determine which fields to omit.

Boost.Serialization takes a library-based approach to providing simple marshalling for C++ datatypes [13]. This library provides largely automatic support for serializing C++ data, but provides no facility for selectively omitting member data depending on context.

Many programming languages explicitly targeted at parallel applications provide automatic marshalling of data or simply present a programming model in which marshalling of user-defined types is not an issue. Generally in programming models where communication is performed via ex-

plicit messages marshalling is not entirely automated in order to allow the programmer some control over how marshalling takes place, but often as in the case of X10 [4] and Chapel [3], the lack of explicit messaging eliminates the need for the programmer to consider marshalling.

6. Conclusions and Future Work

In this paper, we have described a technique for generating marshalling code in parallel applications that improves programmer productivity in several ways. It allows the programmer to specify messages in a way that makes logical sense at the level of application structure, rather than forcing them to construct the minimal set of data required for a given operation and potentially obscuring the relationships between that data. It frees them from having to maintain marshalling routines that are customized to provide some subset of an object's data while leaving extraneous member variables behind, and eliminates the need to update marshalling code when member variables change or functions are rewritten to require different variables. It does this while producing efficient messaging code that excludes member variables which are known to be unusued on the receiving side.

We have implemented this technique using a compiler framework built on top of a popular message-driven runtime system. Applcations built on this programming model communicate via asynchronous remote method invocation between parallel objects. The object-oriented nature of this model harmonizes well with our focus on encapsulation and the preservation of semantic information about the ways in which different parts of an application relate to one another. The actual compiler analysis required to implement our technique is relatively straightforward and relies on well-known, proven techniques for discovering dataflow in an application. We have demonstrated these techniques on two real, albeit simple, applications, highlighting areas in which each application can be improved, either in simplicity or in performance, by applying our technique.

There are many directions this work could be taken in. First, there are some straightforward improvements that have not yet been implemented but are not fundamentally different from what we have presented here. Any method which receives unused data which is stripped out by our technique wastes memory on the receiving side even though no bandwidth is wasted, because full objects are reconstituted on the receiving side rather than just those fields which are necessary. We have not yet implemented this optimization because of time constraints, but it does not present any theoretical difficulties. We could also be more sophisticated in our analysis of arrays and other collections. Currently we do not attempt to detect if only a subset of member elements can ever be used (for example, accessing only odd elements of an array). By making our analysis of which data can potentially be used by a receiver, we could broaden the effectiveness of our technique.

We could also move in several new directions with this work. One major drawback of our approach is that the audience for this technique is limited both by our choice of platform and by our use of a new language and associated compiler infrastructure. Although the language is designed to require minimal porting effort from C++ codes, this is still a large hurdle to overcome. We could produce a separate tool designed to read C or C++ applications and output customized marshalling code. We could also target this work toward MPI applications, looking to extend the work of projects such as Boost.MPI with our technique.

References

[1] J. E. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324, 1986.

[2] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. NAMD: A Portable and Highly Scalable Program for Biomolecular Simulations. Technical Report UIUCDCS-R-2009-3034, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2009.

[3] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21 (3):291–312, 2007. doi: 10.1177/1094342007078442. URL `http://hpc.sagepub.com/content/21/3/291.abstract`.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094852.

[5] CORBA. *The Common Object Request Broker: Architecture and Specification (Draft)*, 10 December 1991. Revision 1.1.

[6] P. Dietz, T. Weigert, and F. Weil. Formal techniques for automatically generating marshalling code from high-level specifications. In *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, pages 40 –47, 1998.

[7] D. D. F. Kjolstad and M. Snir. Bringing the HPC Programmer's IDE into the 21st Century through Refactoring. In *SPLASH 2010 Workshop on Concurrency for the Application Programmer (CAP'10)*. Association for Computing Machinery (ACM), Oct. 2010.

[8] N. Feske. A case study on the cost and benefit of dynamic rpc marshalling for low-level system components. *SIGOPS Oper. Syst. Rev.*, 41:40–48, July 2007. ISSN 0163-5980.

[9] D. Goujon, M. Michel, J. Peeters, and J. Devaney. Automap and autolink tools for communicating complex and dynamic data-structures using mpi. In D. Panda and C. Stunkel, editors, *Network-Based Parallel Computing Communication, Architecture, and Applications*, volume 1362 of *Lecture Notes in Computer Science*, pages 98–109. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64140-

7. URL http://dx.doi.org/10.1007/BFb0052210. 10.1007/BFb0052210.

[10] R. Hillson and M. Iglewski. C++2mpi: a software tool for automatically generating mpi datatypes from c++ classes. In *Parallel Computing in Electrical Engineering, 2000. PAREL 2000. Proceedings. International Conference on*, pages 13 –17, 2000. doi: 10.1109/PCEE.2000.873593.

[11] L. Hochstein and V. R. Basili. An empirical study to compare two parallel programming models. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 114–114, New York, NY, USA, 2006. ACM. ISBN 1-59593-452-9. doi: http://doi.acm.org/10.1145/1148109.1148127. URL http://doi.acm.org/10.1145/1148109.1148127.

[12] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, pages 1–12, 2008.

[13] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar. Modernizing the c++ interface to mpi. In B. Mohr, J. Traff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 266–274. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-39110-4.

[14] J. Kepner. Hpc productivity: An overarching view. *International Journal of High Performance Computing Applications*, 18(4):393–397, Winter 2004. doi: 10.1177/1094342004048533. URL http://hpc.sagepub.com/content/18/4/393.abstract.

[15] L. T. Kou. On live-dead analysis for global data flow problems. *J. ACM*, 24:473–483, July 1977. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322017.322027. URL http://doi.acm.org/10.1145/322017.322027.

[16] T. Panas, D. Quinlan, and R. Vuduc. Tool support for inspecting the code quality of hpc applications. In *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing Applications*, SE-HPC '07, pages 2–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2969-0. doi: http://dx.doi.org/10.1109/SE-HPC.2007.8. URL http://dx.doi.org/10.1109/SE-HPC.2007.8.

[17] C. Queinnec. Marshaling/demarshaling as a compilation/interpretation process. *Parallel Processing Symposium, International*, 0:616, 1999. ISSN 1063-7133.

[18] E. Renault and C. Parrot. Mpi pre-processor: generating mpi derived datatypes from c datatypes automatically. In *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 7 pp. –256, 0-0 2006. doi: 10.1109/ICPPW.2006.56.

[19] J. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[20] M. Snir and D. A. Bader. A framework for measuring supercomputer productivity. *International Journal of High Performance Computing Applications*, 18(4):417–432, Winter 2004. doi: 10.1177/1094342004048535. URL http://hpc.sagepub.com/content/18/4/417.abstract.

[21] T. Sterling. Productivity metrics and models for high performance computing. *International Journal of High Performance Computing Applications*, 18(4):433–440, Winter 2004. doi: 10.1177/1094342004048536. URL http://hpc.sagepub.com/content/18/4/433.abstract.

[22] *Remote Procedure Calls: Protocol Specification*. Sun Microsystems, Inc., Mountain View, Calif., May 1988.

[23] W. Tansey and E. Tilevich. Efficient automated marshaling of c++ data structures for mpi applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, april 2008.

[24] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna. Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers. *Journal of Comptational Chemistry*, 25(16): 2006–2022, Oct. 2004.