

Generating Parallel Applications for Distributed Memory Systems Using Aspects, Components, and Patterns

Purushotham V. Bangalore
Department of Computer and Information
Sciences
CH 130, 1300 University Blvd
Birmingham, AL 35294-1170
1 205 934 8604
puri@cis.uab.edu

ABSTRACT

Developing and debugging parallel programs particularly for distributed memory architectures is still a difficult task. The most popular approach to developing parallel programs for distributed memory architectures requires adding explicit message passing calls into existing sequential programs for data distribution, coordination, and communication. This approach creates separate source tree for sequential code and parallel code as well as further branches based on the specific set of message passing primitives used (point-to-point communication vs. one-sided communication). Aspect oriented programming provides an option to separate programming concerns and weave code into applications instead of directly modifying the original program. This paper described an effort to use Aspect Oriented Programming (specifically AspectC++), components and patterns for data distribution and message passing to develop parallel programs without making any changes to existing sequential program. This technique is used to generate a suite of parallel matrix multiplication algorithms as well as several simple parallel algorithms without making any changes to the sequential code. Performance results obtained indicate that the desired functionality is achieved without compromising performance.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features – *abstract data types, concurrent programming structures, patterns, polymorphism.*

Keywords

Patterns, Parallel Programming, Aspect Oriented Programming.

1. INTRODUCTION

Distributed memory systems (DMS) are currently the preferred architecture for High Performance Computing. Almost all of the 500 machines on the Top 500 Supercomputer Sites list [15] are based on the distributed memory architecture. The most common method used to develop HPC applications is the data parallel paradigm [4] wherein a given dataset is distributed among

multiple processors and each processor is assigned to work on that dataset while exchanging data through explicit message passing. With the data parallel paradigm, data distribution plays an important role in the overall performance of the application because it has impact on load balancing and message passing time [4]. Often, data distribution is determined by the specific application and performed manually by the domain-expert or using automated tools. The next step is to establish communication between different processes. There are several available communication options (such as synchronous/asynchronous, static/dynamic, point-to-point/one-sided/collective), and there are trade-offs with each choice depending on the specific application and underlying hardware configuration. Significant time and effort is involved in refactoring existing code to include message passing constructs and in debugging to ensure correct data is exchanged and synchronization is established.

Typical HPC application development involves adding explicit message passing calls to existing sequential programs. Implicit parallelization approaches using compiler directives (*e.g.*, HPC [7]) and parallel programming languages (*e.g.*, UPC [3]) have not gained significant market penetration compared to explicit parallelization approaches (*e.g.*, MPI [10], PVM [13]). The Message Passing Standard (MPI) [10] – the de facto standard that provides performance portable APIs for message passing – is the most common API used to perform data distribution and setup communication and synchronization between individual processes. The MPI layer deals with explicit buffers and message transfers thereby exposing many of the details about the data structure and data layout and does not provide a sufficiently higher level of abstraction. Furthermore, the MPI function calls will be scattered across multiple source files and HPC application developers have to support different types of communication (*e.g.*, point-to-point vs. one-sided) depending on the underlying architecture. Thus developing, debugging, and maintaining parallel programs is a major challenge since the developer has to deal with all the details of data distribution and data exchange while paying equal attention to performance considerations.

Communication and synchronization are crosscutting concerns in a parallel program that could be handled through the use of Aspect Oriented Programming (AOP) [8] techniques. This paper investigates the use of AOP to weave code into existing sequential programs while using message-passing patterns to provide higher-level abstractions for addressing data distribution and communication. A suite of parallel matrix-matrix multiplication algorithms are generated from an existing sequential program without modifying the sequential program using AspectC++ [12]. The common message-passing pattern in these parallel algorithms is abstracted to provide message passing pattern and this pattern is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

The 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), March 12, 2007, Vancouver, BC, Canada. Copyright 2007 ACM 978-1-59593-657-8/07/03...\$5.00.

weaved into the sequential code using data distribution and message passing components.

Section 2 described how aspect oriented programming is used to generate a simple parallel program using MPI. Section 3 provides an overview of the parallel matrix-matrix multiplication algorithms and the common message-passing pattern seen in these algorithms. Section 4 describes how a parallel version is generated using the message passing patterns and AspectC++. Section 5 describes the experimental setup used, performance results obtained, and performance comparison between hand-written and generated parallel program. The benefits of using AOP for generating parallel programs are discussed in Section 6 and related work is described in Section 7. Conclusions and future work is provided in Section 8.

2. ASPECT ORIENTED PROGRAMMING AND PARALLEL PROGRAMMING

Aspect Oriented Programming (AOP) [8] provides a capability to modularize concerns that are crosscutting in nature and concurrency is one such crosscutting concern that can be addressed using AOP. Most of the current work in applying AOP to parallel computing has been primarily with Java using AspectJ [6][11][14] and there are not many efforts applying AOP to parallel computing using C/C++. But most of the parallel applications are developed using either Fortran or C/C++ and Java is yet to be accepted as a language for high performance computing mainly due to performance considerations and floating point capabilities. The AspectC++ project has provided the C++ community an opportunity to take advantage of the AOP features from C++ and several projects have started to emerge [2]. This is one such effort that leverages the AspectC++ extensions to generate parallel programs for distributed memory systems using MPI message passing APIs.

```
double SumOfSquares(unsigned char *buf, int N) {
    double sum=0;
    for (int i=0; i<N; i++)
        sum += (double)(buf[i]*buf[i]);
    return sum;
}
void Contrast(unsigned char *buf, int N, double rms) {
    for (int i=0; i<N; i++) {
        int val = 2*buf[i] - (int)rms;
        if (val < 0)
            buf[i] = 0;
        else if (val > 255)
            buf[i] = 255;
        else
            buf[i] = val;
    }
}
int main(int argc, char **argv) {
    unsigned char *pixels;
    // initialization code here
    double result = SumOfSquares(pixels, N);
    double rms = sqrt((double)result/(double)N);
    Contrast(pixels, N, rms);
    // finalization code here
    return 0;
}
```

Figure 1. Code segments for a simple image processing program.

In order to understand what is involved in developing a parallel application using MPI for distributed memory systems, consider a simple image processing application that performs a contrast operation on a vector with N elements. Essential parts of the sequential program are shown in Figure 1.

A typical strategy to develop a parallel version of this program is to create multiple processes (say, P) and distribute the vector equally among the P processes. Then each process will compute the partial sum of squares and the partial sum is reduced to a global sum using a collective communication operation (Allreduce). After that each process computes the mean square value and performs the contrast operation. The data is initially distributed among multiple processes and finally collected back after the completion of the contrast operation using collective communication operations (Scatterv and Gatherv respectively). The rest of this section describes how each of above tasks are handled using AOP techniques with AspectC++. For information about AspectC++ please refer to [2][12] for more details.

2.1 Creating multiple processes

While processes creation is typically handled outside of an MPI program, the scope of the MPI environment is defined within two functions MPI_Init and MPI_Finalize [10]. Any code between these two functions will be executed by all the processes created by the MPI runtime system (typically an mpirun script or a daemon process). The functions MPI_Init and MPI_Finalize return when all other processes in the process group have called these functions. Any sequential program can be transformed to a parallel program using the advice code shown in Figure 2. AspectC++ provide the JoinPoint-API runtime methods *Arg<i>::ReferredType *arg<i>()* that returns a typed pointer to the ith argument. These methods are used to obtain addresses of *argc* and *argv* required for the MPI_Init function.

```
aspect mpi {
    int rank, size;
    LinearMapping<int> map;
    // other declarations here
    advice execution("int main(...)"): around() {
        MPI_Init(tjp->arg<0>(),tjp->arg<1>());
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        tjp->proceed();
        MPI_Finalize();
    }
    // other advice code
}
```

Figure 2. Advice code for creating multiple processes.

2.2 Data mapping and distribution

Data mapping and distribution play an important role in the overall execution of the parallel algorithm. If each process is not assigned equal amount of work then the load imbalance will have an impact on the overall execution time [4]. Data distribution and mapping is handled through the use of distribution functions such as linear, scatter, block linear, block scatter, and parametric functions [9]. For simplicity a simple linear distribution function is used within the aspect. This is denoted by the LinearMapping template class in Figure 2. The corresponding *map* object is used by the collective functions to distribute and collect the vector as well as determine the number of elements assigned to each

process. The actual data distribution is performed by the advice around the computation of the sum of squares.

2.3 Sum of squares computation

The sequential program passes the address of the original vector pixels and the total number of elements to the function SumOfSquares. In the parallel program each process receives a part of this vector. The corresponding length of the local vector is provided by the mapping object and the assigned part of the vector is received in a locally allocated vector. The MPI_Scatterv function performs the actual data distribution and these arguments are passed to the original function after saving the arguments passed. Since the function returns partial results, a reduction operation is performed on all the partial results to compute the global sum of squares and this value is returned by the original function.

```

advice call("% SumOfSquares(...)") : around() {
JoinPoint::Arg<0>::Type *buf = tjp->arg<0>();
JoinPoint::Arg<1>::Type *N = tjp->arg<1>();
map.init(*N, size, rank);
counts = map.getCounts();
displs = map.getDisplacements();
mycount = counts[rank];
workbuf = new unsigned char[mycount];

MPI_Scatterv(*buf, counts, displs, MPI_UNSIGNED_CHAR,
workbuf, mycount, MPI_UNSIGNED_CHAR, 0,
MPI_COMM_WORLD);

unsigned char *tmpbuf = *buf;
int oldN = *N;
*buf = workbuf;
*N = mycount;
tjp->proceed();

double *mysum = tjp->result();
double globalsum;
MPI_Allreduce(mysum, &globalsum, 1, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);
*mysum = globalsum;
*buf = tmpbuf;
*N = oldN;
}

```

Figure 3. Advice code for sum of squares computation.

2.4 Contrast operation

Similar to the sum of squares computation the advice code around the contrast operation saves the arguments passed, provides the local buffer and number of elements as the new arguments and invokes the original function. When the original functions returns, the MPI_Gatherv operation collects all the local data pieces and swaps the buffer pointers.

```

advice call("% Contrast(...)") : around() {
JoinPoint::Arg<0>::Type *buf = tjp->arg<0>();
JoinPoint::Arg<1>::Type *N = tjp->arg<1>();
unsigned char *tmpbuf = *buf;
int oldN = *N;
*buf = workbuf;
*N = mycount;
}

```

```

tjp->proceed();

MPI_Gatherv(workbuf, mycount, MPI_UNSIGNED_CHAR,
tmpbuf, counts, displs, MPI_UNSIGNED_CHAR,
0, MPI_COMM_WORLD);

*buf = tmpbuf;
*N = oldN;
}

```

Figure 4. Advice code for contrast operation.

3. PARALLEL MATRIX-MATRIX MULTIPLICATION ALGORITHMS

The common approach to developing parallel algorithms for matrix-matrix multiplication is to distribute the matrices on a two-dimensional process grid, perform multiplication of the local matrix blocks, and exchange the local blocks with other processes using one of the well-established communication patterns [3]. Based on the communication pattern employed and the amount of memory used for storing extra copies of the matrices there are several algorithms documented in the literature and summarized in [3][9]. In this paper three such algorithms [9], namely, broadcast-broadcast, broadcast-multiply-roll (BMR or Fox's algorithm), and Cannon's algorithm are considered and the basic structure of these algorithms is shown in Figures 5-7. The broadcast-broadcast algorithm (Figure 5) uses two broadcast operations to distribute the pieces of matrices A and B. The BMR algorithm (Figure 6) replaces the second broadcast with a shift operation and Cannon's algorithm (Figure 7) replaces the first broadcast with a shift operation, but requires additional shift operations at the beginning.

```

for (k = 0; k < P; k++) {
if (q == k) Broadcast A along processes in the row
if (p == k) Broadcast B along processes in the column
Multiply local blocks A and B and accumulate it in C
}

```

Figure 5. Broadcast-broadcast Algorithm

```

for (k = 0; k < P; k++) {
if (q == (p + k)%Q) Broadcast A along processes in the row
Multiply local blocks A and B and accumulate it in C
Shift/Roll B along processes in the column
}

```

Figure 6. Broadcast-Multiply-Roll Algorithm

```

Shift A p times along process row
Shift B q times along process column
for (k = 0; k < P; k++) {
Multiply local blocks A and B and accumulate it in C
Shift A left along process row
Shift B up along process column
}

```

Figure 7. Cannon's Algorithm

In this discussion it is assumed that the processes are mapped on to a process grid of dimensions P X Q and each process is assigned a coordinate (p, q). Each of the three algorithms described involve the same amount of computation but uses different amount of memory and have different communication complexities. The overall performance of these algorithms is determined by the quality of implementation of the broadcast and

shift operations as well as the problem size [9]. One can observe that there is a common pattern for message passing among the three different versions of the matrix-matrix multiplication algorithm. Figure 8 captures this pattern and this pattern is used with AOP and data distribution and message passing components to generate a suite of parallel matrix-matrix multiplication.

```

Setup()
for (k = 0; k < P; k++) {
    Preprocess()
    Multiply local blocks A and B and accumulate it in C
    Postprocess()
}
Cleanup()

```

Figure 8. Message passing pattern in parallel matrix-matrix multiplication algorithms.

4. GENERATING PARALLEL MATRIX MULTIPLICATION ALGORITHMS

Using the message-passing pattern developed in Section 3 and components for creating a two-dimensional process grid to perform the required communication operations and data distribution an aspect class *mpi* is developed. Figure 9 shows major segments of the aspect base class *mpi*. The components create *2dgrid* and *LinearMapping* handle the creation of two-dimensional process grid and mapping of the problem onto the grid. The advice around *main* creates required number of processes and sets up the MPI environment. The advice around the call to *cblas_dgemm* filters the original arguments and updates these arguments with local variables and incorporates the message passing pattern presented in Figure 8. Note that all the four functions (*setup*, *preprocess*, *postprocess*, and *cleanup*) included are declared as empty functions.

```

aspect mpi {
    int rank, size, P, Q, p, q, myrows, mycols, k;
    int right, left, up, down;
    double *amat, *bmat;
    LinearMapping<int> rowmap, colmap;
    pointcut virtual array_methods() = 0;
    // other declarations
    void setup() {} ;
    void preprocess() {} ;
    void postprocess() {} ;
    void cleanup() {} ;
    advice execution("int main(...)") : around() {
        MPI_Init(tjp->arg<0>(),tjp->arg<1>());
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        // Create a 2-D cartesian topology
        create_2dgrid(MPI_COMM_WORLD,&comm2d,
            &rowcomm, &colcomm, &P, &Q, &p, &q);
        tjp->proceed();
        MPI_Finalize();
    }
    // other advices
    advice call("% cblas_dgemm(...)") : around() {
        // get the arguments
        JoinPoint::Arg<3>::Type *arows = tjp->arg<3>();
        JoinPoint::Arg<4>::Type *bcols = tjp->arg<4>();
        JoinPoint::Arg<5>::Type *acols = tjp->arg<5>();
        JoinPoint::Arg<7>::Type *a = tjp->arg<7>();
    }
}

```

```

JoinPoint::Arg<8>::Type *lda = tjp->arg<8>();
JoinPoint::Arg<9>::Type *b = tjp->arg<9>();
JoinPoint::Arg<10>::Type *ldb = tjp->arg<10>();
JoinPoint::Arg<11>::Type *beta = tjp->arg<11>();
JoinPoint::Arg<13>::Type *ldc = tjp->arg<13>();

*arows = myrows;
*acols = *bcols = mycols;
*lda = *ldb = *ldc = mycols;
*beta = 1.0; // we like to have c = c + a*b not just c = a*b
amat = (double *)*a; // original type is const double *
bmat = (double *)*b; // Sendrecv requires non-const type

setup();
for (k = 0; k < P; k++) {
    preprocess();
    // Multiply local blocks with call to cblas_dgemm
    tjp->proceed();
    postprocess();
}
cleanup();
};

```

Figure 9. Aspect base class *mpi* with advices for functions *main* and *cblas_dgemm*.

This base class is used to derive the three different parallel algorithms described in Section 3. For the broadcast-broadcast algorithm, the two broadcast operations are performed within the *preprocess* function. In case of BMR algorithm, the broadcast is performed within the *preprocess* function and the shift/roll operation is performed within the *postprocess* function. For Cannon's algorithm, the initial shifts are performed within *setup* and the two shift operations are performed within *postprocess* function. Figure 10 provides a snapshot of the derived aspect class for Cannon's algorithm.

```

aspect cannon : public mpi {
    pointcut array_methods() = "% initarray(...)";
    advice execution("void ...:setup(...)") : before() {
        MPI_Cart_shift(rowcomm, 0, -p, &left, &right);
        MPI_Cart_shift(colcomm, 0, -q, &down, &up);
        MPI_Sendrecv_replace(amat,myrows*mycols,
            MPI_DOUBLE, right, 2000,
            left, 2000, rowcomm, &status[0]);
        MPI_Sendrecv_replace(bmat,myrows*mycols,
            MPI_DOUBLE, up, 2000,
            down, 2000, colcomm, &status[1]);
    }
    advice execution("void ...:postprocess(...)") : before() {
        MPI_Cart_shift(rowcomm, 0, -1, &left, &right);
        MPI_Cart_shift(colcomm, 0, -1, &down, &up);
        MPI_Sendrecv_replace(amat,myrows*mycols,
            MPI_DOUBLE, right, 2000,
            left, 2000, rowcomm, &status[0]);
        MPI_Sendrecv_replace(bmat,myrows*mycols,
            MPI_DOUBLE, up, 2000,
            down, 2000, colcomm, &status[1]);
    }
};

```

Figure 10. Derived aspect class for generating Cannon's algorithm.

5. PERFORMANCE ANALYSIS

To compare the performance of generated program with hand-written versions of the same program, both programs were executed on a 128-node cluster with Infiniband interconnect for different number of processors for a fixed problem size. Table 1 provides the time in seconds for generated version and hand-written version of Fox's algorithm for a fixed matrix size of 12000 X 12000 for different number of processors. It can be observed that the generated code performs as well as the hand-written code and did not require any changes to the existing sequential code. Similar performance results were obtained for both broadcast-broadcast and Cannon's algorithm. These results show that the generated parallel code provides similar performance results compared to hand-written code, which is error prone and hard to debug. With the use of appropriate components and patterns one can weave these components and patterns into sequential code and develop parallel applications without requiring modifications to the sequential code base.

Table 1. Comparison of execution times for Fox's algorithm

No. of Processors	Generated Code Time (seconds)	Hand-written Code Time (seconds)
1	716.743161	712.142265
4	161.233066	161.489131
9	73.213474	73.163756
16	42.767262	42.755549
25	28.518635	28.443467
36	20.234452	20.098236
64	11.898304	11.710446
100	7.963589	7.848395

6. DISCUSSION

The performance analysis of the generated code and hand-written code revealed that there was no significant performance degradation from using AspectC++ to generate the parallel code. In addition, the use of AOP enabled separation of the message passing code from the original sequential program as well as eliminated the need to modify the sequential program and create a separate parallel version in the source tree. The aspect advice to create multiple processes (shown in Figure 2) was used in all MPI programs generated with AspectC++. The aspect advice for allocating matrices and initializing matrices was reused through aspect inheritance and only the message passing primitives specific to the algorithm were included in the derived aspect class. The base aspect class could be further reused for any matrix related application. If one were to replace the point-to-point communication calls with one-sided calls or replace the collective communication calls with specialized point-to-point implementations, this could be easily accomplished by extending the aspect base class *mpi*. The message-passing pattern described in Figure 8 avoided the use of virtual functions in the derived aspect classes. The empty functions (setup, preprocess, postprocess, and cleanup) were used as placeholders to weave in code specific to the algorithm as shown in Figure 10.

Traditional explicit parallel programming techniques require manual insertion of MPI or PVM function calls or other libraries written in MPI or PVM. Using AOP techniques we can automate these code changes and isolate the changes within aspect advice. The aspect advice would further take advantage of data

distribution, communication components, and patterns inherent in the application. These components and patterns could be designed and developed by users with expertise in parallel computing and the advice writer (who may not be an expert in parallel computing) could use these components to generate the parallel program, thereby separating data distribution and communication concerns for the parallel program developer.

7. RELATED WORK

Most of the work to date in developing parallel applications using AOP has been primarily in Java [6][11][14]. In [6] object-oriented models to capture loop-level parallelism is provided along with experiments on legacy scientific applications in Java. A paradigm for developing parallel applications in a step-wise fashion is described in [11]. Design patterns for parallel programming in Java is provided in [14]. But these techniques cannot be directly applied to generating parallel programs in C/C++ since the AOP support for C/C++ is very limited at this time.

With respect to AOP support for C/C++, AspectC [1] and AspectC++ [12] are the two popular options and they have been used in middleware development, operating system design, and real-time applications [2]. However this is the first time AOP technique using C/C++ have been applied to generate parallel programs.

8. CONCLUSIONS AND FUTURE WORK

This paper described an approach to generate parallel programs by combining Aspect Oriented Programming techniques with data distribution and message passing components and patterns. Using this approach a suite of parallel matrix-matrix multiplication algorithms was generated and the performance of these algorithms was compared with similar versions of hand-written MPI programs. Performance results obtained illustrated that the generated code performed as well as hand-written code. While the initial performance results are very encouraging this approach needs to be expanded to other parallel algorithms and performance implications need to be further investigated. This is the first step in that direction.

9. ACKNOWLEDGMENTS

The author would like to thank the Department of Computer and Information Sciences at the University of Alabama at Birmingham and the National Science Foundation grant CISE-MRI-0420614 that provided the resources for testing the parallel programs presented in this paper. Special thanks are also due to my colleagues Dr. Anthony Skjellum and Dr. Jeff Gray for their support and encouragement for this work.

10. REFERENCES

- [1] AspectC Homepage. <http://www.aspectc.net>.
- [2] AspectC++ Homepage. <http://www.aspectc.org>.
- [3] El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K., *UPC: Distributed Shared Memory Programming*, Wiley, June 2005.
- [4] Grama, A., Gupta, A., Karypis, G., Kumar, V., *Introduction to Parallel Computing*, 2nd Edition, Addison Wesley, 2003.
- [5] Gunnels, J., Lin, C., Morrow, G., and van de Geijn, R., *A Flexible Class of Parallel Matrix Multiplication Algorithms*,

- Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98), pp. 110-116 1998.
- [6] Harbulot, B., Gurd, J., *Using AspectJ to separate concerns in parallel scientific Java code*, Proceedings of the 3rd International Conference on Aspect-oriented Software Development, Lancaster, UK, pp. 122–131, 2004.
- [7] High Performance Fortran Forum, *High Performance Fortran Language Specification, Scientific Programming* 2(1-2), pp. 1-170, 1993.
- [8] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J. and Irwin, J., *Aspect-Oriented Programming*, European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, Jyväskylä, Finland, June 1997, pp. 220-242.
- [9] Li, J., Skjellum, A., Falgout, R. D., *A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. Concurrency: Practice and Experience*, 9(5), pp. 345--389, 1997.
- [10] Message Passing Interface Forum, "MPI2: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications and High Performance Computing*, Special Issue, 12(1/2), pp. 1-299, 1998.
- [11] Sobral, J. *Incrementally Developing Parallel Applications with AspectJ*, 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS'06), Greece, Rhodes, April 2006, IEEE Computer Society, 2006.
- [12] Spinczyk, O., Lohmann, D., Urban, M., AspectC++: an AOP Extension for C++", in *Software Developer's Journal*, pp. 68-76, 2005. Sunderam, V. S., *PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience*, 2(4), pp. 315--339, 1990.
- [13] Sunderam, V. S., *PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience*, 2(4), pp. 315--339, December, 1990.
- [14] Tan, K., Szafron, D., Schaeffer, J., Anvik, J. MacDonald, S., *Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment*, PPOPP'03, San Diego, California, USA, June 2003.
- [15] TOP500 Supercomputer Sites. <http://www.top500.org>.