ParaC: A New Approach to Practical Parallel Programming

Christopher J. Hazard

Advisor: James Caristi

November, 2001
Abstract

ParaC is a distributed programming language based on C++ and designed to help make distributed computing across a cluster of networked workstations easier to develop. It allows the programmer to describe what should be computed in parallel, but shields the programmer from the nuisances of network communication and management. ParaC’s design is optimized for real world topologies, and provides the programmer with a great deal of flexibility and development power.
# Table of Contents

1. Introduction

2. Background
   2.1 History and Driving Force for Parallel Computing
   2.2 Current Technology
   2.3 Difficulties in Distributed Computing

3. A New Approach
   3.1 Midway Between the Extremes: A New Language
   3.2 Language Constructs
   3.3 Benefits of the New Approach

4. ParaC Architecture
   4.1 Overview
   4.2 Compiler Front-End
   4.3 Compiler Back-End
   4.4 Communications Interface
   4.5 Communications Control

5. Results and Future of Project
   5.1 Test Cases
   5.2 Planned Additions to Language and Architecture
   5.3 Impact on Parallel Computing

6. Conclusion

7. Bibliography
1. Introduction

Despite the fact that computers have been doubling their speed every 18 months for much of the history of computing, there are always numerous applications, simulations, and scientific calculations that require a faster machine. Not only are there more and more problems to solve that require the fastest computers in the world, but also there are becoming an increasingly large number of problems that just require something beyond what high-end workstations can do.

The use of parallel and distributing computing has always been an answer to the question of more computing power. It sounds like a great answer, but the practical applications of distributed computing have always been restricted because of complexity of development.

With each step of the way, parallel computing has been becoming more practical and easier to develop. In this paper, I present a novel approach to development of distributed computing. Most current technologies either require a great deal of effort be placed into the development of the intricacies of parallelism, or put the burden completely on the compiler, creating a less efficient program. My approach is to choose a medium ground between the two, allowing the programmer to have control over how the program becomes parallelized, but not have to deal with the complexities of network communication and load distribution.
2. Background

2.1 History and Driving Force for Parallel Computing

As every generation of computing speeds is released, a new generation of problems arises that requires yet more computational power. In order to try to leap ahead of the technological barriers of faster computational equipment, computers have been linked in parallel in attempt to create a computer with more simultaneous processing power.

Parallel computing began in the late 1950’s by having multiple processors connected together, which included instructions for the processors to wait for the other one to finish. This technology evolved into connecting larger numbers of processors together, and remained prevalent in only the fastest computers produced.

Slowly, multi-processor computers took grip in more areas of computing, such as Internet servers and powerful workstations. As the operating systems have been able to better take advantage of the parallel processors, they have become more common.

Still, these multi-processor computers only have about two processors, with a few exceptions that range up to eight processors. The super computers are the only machines that have large numbers of processors.

Because of the prevalence of networks and the Internet, network equipment has become more common. With the inexpensiveness of workstations and network equipment, it is easy to create a large network of fast machines, which are cheaper and potentially more powerful (for most purposes) than super computers.

Historically, super computers have only been used for large scientific calculations, not only because of price but also because of development complexity.
Now that there are becoming more uses for extra powerful computers (meaning computers more powerful than current commercial or consumer grade computing equipment), and the price of distributed networks has dropped, only the development complexity remains.

The development complexity is still an issue because the programmer must not only deal with the problem of developing the program, but developing it with parallel algorithms, in addition to all of the network communications and synchronization.

### 2.2 Current Technology

The three most widely used methods for parallel computing are parallel execution units in processors, multi-processor machines and clusters of workstations. Each technology has its own strengths and weaknesses, are used additively to gain the benefits of each.

Parallel execution units have been one of the largest driving factors in computing speed throughout the 1990’s. Multiple copies of the execution units on a processor are controlled by complex mechanisms that rearrange the instructions so that the maximum can be computed simultaneously. These mechanisms include superscalar architecture, pipelining, and out-of-order execution. They parallelize at a very low level, and boost the processor computational power tremendously. These parallelizations, however, are often attributed to the speed of the processor, because they are transparent to the development, and in all but rare cases, are transparent to the program itself.

Multi-processor architectures often have a set of $2^n$ processors with some or all of the memory shared between them. These, machines have slowly gained strength and support, as operating systems have been better able to use them, but are not as scalable,
as the communications and synchronizations between the processors becomes more complex with every processor added. They are able to take advantage of things such as shared memory and multi-threading across processors.

Most supercomputer architectures are somewhere between multi-processor architectures and clusters of workstations. Their busses are often designed like a network, with each processor having its own board to take care of its communications, in addition to hosting its own memory.

Even with the multi-processor machines, it is usually much more economical to have many high-end workstations than a single supercomputer. This is the emergent technology of clusters of workstations. They have slowly become more popular as research has made it easier to utilize the computational power. A cluster of workstations is simply a number of workstations connected via a network. To utilize the computing power, complex overhead of message passing, network management, and resource utilization must have to be dealt with. This includes data and control transfer, and also has to take into account things like load balancing (it would be very inefficient to have a large network where only two computers are doing all the work).

To address the problems of network communication, several standards and libraries are used. Most of the distributed computing has been on the Linux platform because of its free availability and robustness. The most widely used libraries are straight network programming, PVM (Parallel Virtual Machine), MPI (Message Passing Interface), and AFAPI (Aggregate Function API), with some of the main projects being Beowulf, Mosix, NOW (Network Of Workstations), and Linux/AP+. The libraries and
protocols vary at different degrees of complexity versus low-level flexibility, but all require complex network programming.

2.3 Difficulties in Distributed Computing

Many very difficult issues arise in parallel computing across a cluster of workstations. Each of the workstations knows how to talk to any other workstation, but all of the communication itself must be established. There are packages such as PVM and MPI that do most of the low-level network communications, but still require the programmer to go to great lengths to build in the parallelism, and do not allow for much flexibility. In addition, these libraries are relatively complex to learn, and in some ways difficult to use.

For example, the code must describe how the node relates to the computation, and what role it plays. This means that the node must recognize what it is and do the corresponding communication and computation.

On the other end of the spectrum are compilers that try to do all of the parallelizing automatically, such as done by the Portland Group (Internet site: http://www.pgroup.com). In these cases, all of the parallelization is taken care of by automation. The compilers use straight C++ or FORTRAN code and attempt to pull out the portions of the program that are best suited to be run across different machines. This makes it very easy to develop for, but rarely comes close to the efficiency of programs written for distributed computing. This is because current compiler technology cannot see a program from high enough level to “understand” the program well enough to optimize it for distributed computing. What the compiler does do is try to unravel loops and call out for other nodes to execute the loop body (which can include function calls).
Ultimately, the compiler would take care of doing all of the optimizations and parallelizations, but it will be a while before compiler technology will be able to do that.
3.  A New Approach

3.1  Midway Between the Extremes: A New Language

Given the two main distributed programming methodologies (writing the networking and communication code manually versus writing a standard language and having the compiler attempt to optimize it) almost nothing exists between these two extremes. They both have great advantages for specific cases, but the general case is still the largest hurdle. Using a distributed cluster to solve a variety of problems easily and efficiently is one of the primary reasons that clusters are not the used in nearly as many situations as they might be. As development for clusters has become easier, clusters have become increasingly more popular.

My approach was to give the programmer some control over how the program is parallelized, yet shield the programmer from all the nuisances of network and communication control. The idea behind this is to allow the programmer to tell the compiler that this part should be parallelized, but let the compiler and communications code decide how to parallelize it. This is realized in my parallel programming language, ParaC.

3.2  Language Constructs

ParaC is essentially a superset of C++, meaning that it contains all of C++’s constructs and styles, in addition to containing other types of constructs. In my implementation, there are parts of C++ that are not implemented, such as class inheritance, templates, and pointers. Class inheritance could be implemented on a future version, but templates and pointers were omitted purposely to allow for cleaner and more efficient handling of parallelizing code. Pointers were left out more specifically because
nodes would not have access to other nodes’ memory. In order to allow access to other nodes’ allocated memory, a great deal of memory overhead would need to be put into place. This would require allocated memory to be sent from node to node constantly, which would slow the system down, in addition to creating synchronization bottlenecks.

Memory allocation and deallocation is dealt with by declaring arrays of size zero. When an array is declared of this size, it is allowed to be resized during runtime. To allocate or deallocate memory, the array is resized with the := operator, where the right side contains the new length. The communications stub manages the memory allocation.

3.2.1 Parallel Classes

The two major new constructs are parallel classes and parallel blocks. Parallel classes (denoted by pclass) are classes that are stored on a node predetermined by the communications stub (this will be explained in the architecture section). All attributes of a parallel class are given private status, meaning that nothing outside the class can access them, thus all variables must be accessed by accessor functions. This makes it easy for the compiler to keep the parallel class on another machine and access it efficiently. This could be hidden from the programmer by having the compiler create accessor functions, but I decided that it would be good for the programmer to keep in mind whether or not a class is a parallel class or not, so that the programmer will produce more optimal code (versus continually accessing a parallel class attribute).

If a parallel class member function call returns a value, the caller will wait for it to return. If it does not return a value, then it does not wait, but continues on, with both machines doing things in parallel. Since the parallel class resides on one machine, when it receives its next member function call, it will wait for the previous one to finish (as
each node by definition can only execute one thing at a time—it is not restricted to one node per machine as different processes can be created, as this is very useful for multi-processor systems).

### 3.2.2 Parallel Blocks

The heart of the parallel architecture, however, is the parallel blocks (denoted by @’s). The blocks cannot be nested, and indicate that anything in them should be parallelized if at all possible. Parallel blocks are best thought of to execute all lines in the parallel block simultaneously (even though this is not the case, but thinking this way allows the programmer to use them most effectively), and they have some restrictions. The first is that no new variables may be declared within a parallel block because there is a chance that they could be used before they are initialized. The second restriction is that no program control statements besides function calls are allowed within the block (i.e. if, else, while, for). This is because, by definition, all lines are treated like they are executed simultaneously.

All function calls within a parallel block are executed in parallel on other nodes, which is done upon entry of the parallel block. The code to obtain the parameters for the functions is executed first; then the functions are called. This is the reason that it is best to think that all the statements are executed in parallel, because the code that obtains the parameters is executed out of order. Once the parallel calls are made, the block is run, and at each function call, it waits for the return value. If the returns come back out of order, it makes for virtually no speed loss, because they are stored in a list, and out of order returns will not have to be waited for when they are actually needed in the code. The power of this is that the functions called in parallel may also contain parallel blocks.
to call other functions. This is how the tree-like structure of parallel processing nodes is traversed depth-wise. For parallelizing loops, parallel blocks are placed within the loop, and it is unrolled a ‘little bit’. A ‘little bit’ depends on the topology, for example a hypercube topology with 16 nodes would optimally be unrolled by a factor of 4 (because each node would have 4 neighbors: $2^4=16$), but as long as the loop is unrolled by some number in the neighborhood of the logarithm of the number of nodes, then it will work well.

3.3 Benefits of the New Approach

Because of the parallel blocks, ParaC is able to traverse the topology of the network efficiently, and is easily adaptable to virtually any type of topology. It is also very easily optimized for topologies where the number of edges (connections) is proportional to the logarithm of the total number of nodes. This is very significant because this is generally the type of hardware that can be engineered economically. In addition, the topology for the parallel network can be defined by the actual physical layout and connections of the hardware. ParaC has a network topology file that specifies each of the node addresses, and each of the edges (connections), and a program could be written to check the latencies and bandwidth of various network connections and generate a topology that would be optimal (or nearly optimal) for the given physical hardware layout. Again, this is a very big implication, as it would allow users to create networks that best suited their needs, and not have to worry about whether or not it will be difficult or inefficient to run parallel programs on (however, if the topology is very bad, which is unlikely because of general network design, the distributed programs would suffer lower performance).
ParaC will also run on any machine that supports a POSIX compliant operating system that includes BSD sockets. The current version only supports i86 processor architecture, as this is the most prevalent architecture, but it is easily extendable to other architectures. Because of this, ParaC can operate on inhomogeneous networks (consisting of machines with different speeds and memory), as the load balancing mechanism sends parallel jobs (function calls) to the connected node with the fewest number of jobs left to do.

Furthermore, ParaC’s network control programs allow easy parallel program administration. All of the I/O (Input and Output) to and from the program is done through the program that starts the parallel program. It can be stopped and restarted with very simple commands. In addition, it allows file management for all of the nodes, treating them all the same, so they each have the same files. This can be useful for an input file, as you could have different nodes read different sections of the same file or files.

As expected from any system-like programming language, ParaC supports linking to external data and external functions. It supports linking to any library file, with the exception that ParaC does not allow pointer manipulation. This can be useful for a large variety of reasons, such as using optimized libraries, or accessing hardware or external I/O systems. An example of this would be to have a simulator running written in ParaC, which communicates to a server that controls robotics. This makes ParaC a very flexible language.

To illustrate an example of how easy it is to create a trivial solid parallel program in ParaC, here is the listing of a program that counts the number of factors of a number,
by dividing the task in half across two nodes (it could easily be divided up more, which
would be more beneficial for larger clusters):

Listing 3.3-1
#include con_io.h

long findnumfactors(long number, long begin, long end)
{
    long i;
    long count = 0;
    for(i = begin; i <= end; i = i + 1)
    {
        // see if remainder (modulus) left after division
        // if not, then it is a factor
        if(number % i == 0)
            count = count + 1;
    }
    return count;
}

void main()
{
    // read in a number from the console
    long number = con_get_integer();

    int a = 0;
    int b = 0;
    @
    // these are computed in parallel
    // a finds the number of factors in the range of
    // 1->number/4 and b finds factors between
    // number/4->number/2
    // by definition, a number cannot have a factor
    // greater than the number divided by 2
    a = findnumfactors(number, 1, number / 4);
    b = findnumfactors(number, number / 4 + 1,
        number/2);
    @
    // print out result
    con_write_integer(a+b);
}

To run this program, at the main console, you would type:
parac numfactors.pcc
./launch.pl numfactors
That’s all there is to it. The first line compiles the program, and the second one runs it
across the distributed network.
4. ParaC Architecture

4.1 Overview

ParaC consists of four main sections. The first section is the ParaC language compiler, which takes in the source code, and writes out i86 assembly. This is divided into two sections: the front end that reads in the code and converts it into an intermediary form, and the back end that optimizes and converts the intermediary form into i86 assembly. The communications interface is linked to the output of the ParaC language compiler, to provide the network access and control over the program. Finally, the communications control is the user interface to launch and control the distributed network.

4.2 Compiler Front-End
The first portion of ParaC’s architecture is the compiler front end, which reads in the ParaC source code and transforms it into an intermediate form. This extra intermediate form makes it easier for optimizers to be implemented (which optimize the resultant program for execution speed), as well as moving the code closer to its final form. As typical of most compilers, it contains a symbol table, string table, scanner, and a parser.

The string table contains all of the identifiers from the ParaC source, such as variable names, function names, reserved words, etc. When a string is encountered, its token is looked up. Each unique string has a unique integer token, which allows the compiler to easily manipulate identifiers, as opposed to doing large amounts of string manipulation. ParaC keeps the strings in a trie structure to allow for $O(n)$ access time with respect to the length of the string.

The other main data structure in the compiler is the symbol table. The symbol table keeps track of everything related to any identifier, such as data type, size, memory location, scope, in addition to the syntax trees of the functions. Nearly all of the data used to represent the source code in the compiler is stored in the symbol table.

When the compiler is invoked, the source code input must be first translated into a form that is easier for the processor to work with. This action is done by the scanner, often referred to as a tokenizer because it turns the source code into tokens. Tokens are internal symbols that represent the source code, but are easier to work with for translation. It uses the string table to translate identifiers into tokens, and translates all other symbols, including operators and numbers, into their respective tokens. The scanner also takes care of included files, and does some first tier error checking.
The parser is the main portion of the front end of the compiler. It pulls in the tokens representing the source file from the scanner and transforms it into an abstract syntax tree. The parser keeps track of scope and applies the language’s grammar to its input. It begins assuming global scope, reading in variables, class definitions, prototypes, and functions, adding them to the symbol table. When the parser comes across a function, it parses the function into an abstract syntax tree. The abstract syntax tree is a preorder binary tree that contains the statements of the function. It builds this tree by putting the language constructs and statements in the tree, and by using a stack-based system along with its precedence rules for the expressions (expressions defined here as subsets of statements).

The end result of the front end of the compiler is the symbol table containing all of the information about the target program. Within the symbol table’s entries lie the abstract syntax trees, which define the contents of the functions. Now the source code is in an intermediate form that explicitly defines all aspects of the target program.

### 4.3 Compiler Back-End

![Diagram of Compiler Back-End](Image)
The intermediate representation produced by the compiler front end is now ready to manipulate. The compiler’s back end optimizes the program for speed and writes it out to the native machine code. In the case of ParaC, the compiler writes it out to assembly language, which is a human-readable representation of the native machine code, which can then be assembled and linked. The reason for writing it out to assembly versus machine code is that it makes the program more portable because of the vast number of library formats and executable formats.

ParaC’s back end, like many modern compilers, translates the abstract syntax tree into one more intermediate representation before finally translating it into assembly language. The benefits of this extra step are two-fold. It makes the backend more modular, thus easier to maintain and easier to migrate to different platforms in addition (as only the last portion of the backend must be replaced by a module that translates to a different assembly or machine language). It also makes room for additional optimizations that could otherwise not be done (given current compiler technology).

The abstract syntax trees from the front end are passed to a medium level code generator. The abstract syntax trees are traversed, concatenating medium level instructions together and extracting information from the symbol table that is required in the intermediary representation (such as pointers and address calculations). The medium level code generator pulls apart the operations into triplet instructions (i.e. \( c \leftarrow a + b \) where \( c \) receives the value of \( a + b \)), and keeps track of temporary values for arithmetic operations involving more than one operand.

The medium level representation is essentially a very simple programming language. It can be thought of as existing somewhere between a high-level language and
assembly language. Some of its triplet instructions resemble instructions for various architectures (such as MIPS), but with many fewer restrictions.

Listing 4.3-1 is the unoptimized medium level representation for listing 3.3-1, and includes a variety of language features such as parallel calls. It should also be noted that identifiers are listed as the identifier name followed by a comma, followed by their definition number (which is needed for repeated use of identifier names in different scopes, i.e. a global variable having the same name as a local variable or class attribute). Temporaries are listed by the character sequence ‘t’ followed by a number.

**Listing 4.3-1**

```
sub: main
    call getnumber,1
    pull t9
    number,2 <- t9 + 0
    a,1 <- 0 + 0
    b,1 <- 0 + 0
    t6 <- number,2 / 4
    push t6
    push 1
    push number,2
    pcall findnumfactors,1 wait_id: 2
    t5 <- number,2 / 2
    t3 <- number,2 / 4
    t4 <- t3 + 1
    push t5
    push t4
    push number,2
    pcall findnumfactors,1 wait_id: 1
    waitfor findnumfactors,1 wait_id: 2
    pull t8
    a,1 <- t8 + 0
    waitfor findnumfactors,1 wait_id: 1
    pull t7
    b,1 <- t7 + 0
    t1 <- a,1 + b,1
    push t1
    call putchar,2
    pull t2

sub: findnumfactors
    count,1 <- 0 + 0
    i,3 <- begin,1 + 0
    1:
```
t1 <- i,3 <= end,1
if t1 != 0 goto 2
  t3 <- number,1 % i,3
  t4 <- t3 == 0
  if t4 == 0 goto 3
  t5 <- count,1 + 1
  count,1 <- t5 + 0
  goto 4
3:
4:
  t2 <- i,3 + 1
  i,3 <- t2 + 0
  goto 1
2:
  return count,1

The results of the medium level translation are passed to a medium level optimizer. Medium level optimizers perform operations such as algebraic simplification, various loop optimizations such as variable induction and strength reduction, redundancy elimination to mention a few. ParaC currently only performs constant folding (combining algebraic manipulations of constants), but other modules could be easily expanded to support other optimizations.

Once the medium level representation is optimized, it is passed to the code generator. Besides some final optimizations such as peephole optimization, this is the final step in the compiler. The code generator converts the medium level representation into the actual instructions the processor will execute to run the target program. Because the target platform is i86 (the primary architecture of PCs), 109 of the most widely used instructions are used (as the backwards compatibility and future expansion of PC instructions have many non-standard or nearly deprecated instructions). This does not include floating point instructions, which are not generated by the current version of ParaC (omitted because of time constraints).
Much of the code generator’s job is to translate the medium level instructions to the corresponding sequence of low-level instructions. Since this is the final translation of the target program, all of the necessary information stored within the symbol table must be put into the resulting output. This includes addresses, sizes, and type conversions. The main difficult portion of code generator’s job is to perform register allocation. Registers are the temporary containers of data that reside on the processor itself. This processor architectural practice of using registers is used in nearly all processors, because of the benefits gained from it. But because of this architecture, in order to perform an arithmetic computation, values must be loaded from memory into registers, and then stored back out when the computation is done. Most processors have only a relatively small number of registers (newer architectures may have as many as 1024 registers, but they will not be widespread for some time –yet even these new architectures will need register allocation, because there is a finite number of registers), and because all calculations must be done with registers, careful and efficient manipulation of data into and out of registers is required. There are many common methods of register allocation, the most popular being graph coloring, which usually works best for academic and high performance style architectures. The relatively exotic yet ubiquitous i86 architecture was not designed for academic purposes and was not originally designed for high performance (although relatively high performance is currently obtained because of advances in computer architecture), I decided chose to create my own method of register allocation for ParaC.

Register allocation is usually performed on medium level representations that try to minimize the number of temporary values needed. I decided to take a different
approach, and build a register allocator designed around a medium level translator that
does not minimize the number of temporaries, and designed the medium level translator
as such. My method of register allocation is to cycle through the registers and push a
register onto the stack when needed for each temporary. Future references to this value
will try to pull it from the top of the stack, or merely retrieve the value by the address of
the saved value in the stack. After each medium level instruction, the register allocator
checks to see if any of the temporaries previously used are ever used again. If a
temporary is not used again, it cleans up the stack, removing values in front of older
values that will eventually be reused, increasing their chances of being able to be popped
off the stack into an empty register. In addition, it checks to see if any registers contain
data no longer used, and flags them as unused if they do.

Finally, the code generator must also create all of the exporting, size information,
and alignment for functions and global variables. In addition, because this is a distributed
programming language, it must contain information for the communications interface, to
allow for remote function calls. To do this, it enumerates all of the functions, and treats
each function by its enumerated value looked up in a global table.

### 4.4 Communications Interface

![Diagram of Communications Interface]

program to allow for network communications and management. It can be thought of as
the interface between the target program and the parallel copies of itself. It has two stages of execution: initialization and synchronization, and dispatching jobs.

When the distributed program is run on each of the nodes, the communications interface is executed before the target program (actually, the communication interface in a sense executes the jobs that make up the target program). The communications interface sets up all of its communications servers to listen for new connections. It listens on the ports specified by the topology file, which is an initialization file that specifies the topology of the distributed computer, as the topology is completely configurable. Once it is ready to listen for connections, it sends a message to the launcher, which is part of the communications control, stating that it is ready to begin connecting. The launcher waits for all of the parallel modules to reach the ready state, and then sends a message to all of them to tell them to connect. The communications interface on each of the parallel modules proceeds to connect to all of its neighbors as specified by the topology file. Once it is completely connected to all of its neighbors, it sends a message back to the launcher stating that it is ready to begin execution jobs. Once the launcher receives this message from all of the nodes, the distributed program is ready to begin.

The launcher then sends a message to the first node to tell it to add the main function of the distributed program to the jobs list. The communications interface consists of a job execution queue, and a return value list. It has threads that monitor each of the connections looking for messages for new jobs or job return values, and stores each in the respective data structure. The primary thread spawns several new threads to processes the job queue, calling the respective function from the distributed program with the corresponding parameters. When the distributed program comes across a parallel
function call, it calls a routine from the communications interface that dispatches the job to another node. Load balancing is implemented by the communications interface polling each of its connections for the one with the shortest job queue, and dispatching the job to that node. Once a job is finished, it is removed from the queue and if a return value is requested, it is sent back to its calling node.

While a job is waiting for a return value, it continually polls the return value list to see if it has received it. This allows the other threads to accept returns from other nodes while waiting for the current return value, and allows other jobs to use this “free time”. If a return value is already in the list (i.e. it was sent back before the job needed it), then it is received by the job immediately.

The communications interface also contains a small library of console input and output routines that communicate with the launcher, thus providing all console input and output from a single console.

4.5 Communications Control

Because the purpose of ParaC is to make development of distributed programs easier, it is essential that the developer and user have the ability to easily execute, manage, and terminate the program. Since these network management tools are
independent of the compiled target program, and are only to start and stop the program, their efficiency is not as critical. I chose to do this in the language Perl because of its rapid development features and its wide availability. The communications control is essential for coordination of the distributed program’s initialization across the network, and gives the user a very easy interface. The communications control is divided into two parts, the node server and the user commands.

The node server is run on every node in the distributed network, where a node is defined as a process and set of ports on a machine on the network. An individual machine may host several nodes, (this is only beneficial in two circumstances: multi-processor machines, and some types of coarse-grained programs that could weaken the effect of load balancing) but it is usually best suited to have one node per machine. The node server allows user command programs to connect to it to send it commands, including commands like, launch distributed program, retrieve file, and terminate node. The node server will launch a program, and keep track of its process in order to kill the process if it receives a command to do so. It can also ftp into a remote machine to retrieve a specified file, which is useful for allowing all nodes access various positions of an input file simultaneously and independently.

The user commands are Perl programs that communicate with the node servers and the running distributed program. It includes commands to send a file, stop all nodes’ programs, and launch a distributed program across the network. Sending a file results in all nodes having a copy of the same file, and stopping all nodes’ programs halts execution of the distributed program and frees up consumed network resources. Launching a program is the most complex, as it also communicates with the distributed program on
every node. When launching a distributed program, it sends a message to all of the node
servers to start the distributed program, waits for a response from all of the launched
programs, and communicates with all of them to synchronize them before the actual
distributed program begins execution. Once all of the launched programs are ready, it
initiates the main function of the program on the first node. The launcher also keeps
connections open to all of the running distributed programs, to pull all console input and
output through the launcher (so that all of the user input and output are done on the
machine from which it was run).
5. Results and Future of Project

5.1 Test Cases

ParaC was tested thoroughly throughout the development process, but as any larger scale programming project, is prone to have bugs that will have to be found merely by usage and time. As of writing this paper, ParaC was tested mostly using trivial example programs, but was also tested with the factor counting program from listing 3.3-1, and has been shown to distribute the execution across the distributed network. More complex tests are being developed, and should be complete within a couple of weeks.

5.2 Planned Additions to Language and Architecture

I plan to continue this project after graduation because I see many potential benefits from it. My language was designed with the intent of making a production grade environment: very complete, robust, useful and practical. As this project had the time scope of one person-year, there are many language features that would have been impossible to complete in that time, but I have left room in ParaC for them. Here, I will describe the language features that have not yet been implemented as of the time this paper was written.

The first, and probably most important feature to be added is floating point support. This is handled all the way up to the code generator, but was left out of the code generator because of the time required to implement a code generator for i86’s complex floating-point instructions. This feature is essential for use in almost any type of scientific or engineering calculation.

Dynamic memory allocation and deallocation has become essential to large and even many smaller scale programs. This feature is almost completely implemented in
ParaC, but a memory management module must yet be added to the communications interface to allow it to work.

Although ParaC supports classes, it does not support class inheritance. The ParaC compiler architecture is very supportive to class inheritance, including advanced class inheritance technologies such as virtual functions.

One of C++’s features is that many of its operators combine to make shorter and more concise expressions, such as the +=, *=, ?:, and ++ operators. This requires only small modifications to the parser.

ParaC’s optimizer currently only does constant folding, which is algebraic combining of constants. Many other optimizations could be done, which would greatly enhance performance.

Because the effectiveness of load balancing is dependent upon how the program is coded, load-balancing statistics would be very useful. They would provide the programmer with information similar to a profiler, but instead with statistics on idle time for each of the nodes. This would be very useful in optimizing code to take advantage of the distributed network.

5.3 Impact on Parallel Computing

This project has demonstrated a new, efficient, and easy method of distributing a program across a cluster of workstations. This is significant, as this problem is currently the largest problem related to distributed computing. ParaC provides a powerful method of tapping into the computational power of distributed networks.

The primary feature of ParaC is its parallel blocks. Because the programmer specifies what is done in parallel, the programmer can optimize for the case of the
number of connections a node has, without worrying about how it is done in parallel. This optimizes for current distributed computing hardware, where the number of connections is typically on the order of the logarithm of the number of nodes.

Another interesting research project would be to learn more about the register allocator I invented. There are many different strategies of register allocation, and it would be interesting to find out what benefits and drawbacks my method has.

Perhaps the final impact on parallel computing is ParaC’s ease of use. Installation is merely copying ParaC to all of the nodes of a distributed network, specifying the connections, and specifying the topology. New machines can be added or removed to or from the network easily, and the ParaC server running on each machine can be used to distribute the new topology in a single command. Compilation and execution of a parallel program only require one simple command each. Development of a parallel program only requires specifying what should be parallel.
6. Conclusion

ParaC’s innovative approach to development of distributed computing programs is a forward step to allow distributed networks to become more usable. Distributed networks are a very large, still untapped resource for supercomputing. By allowing easy yet efficient development, ParaC could be used as a powerful tool to aid and advance research and use of distributed networks.
7. Bibliography


