FastPara: a High-level Declarative Data-Parallel Programming Framework on Clusters

Yong Mao, Yunhong Gu, Jia Chen, and Robert L. Grossman
Laboratory for Advanced Computing, University of Illinois at Chicago
322 SEO,M/C 249,851S.Morgan,Chicago,IL.60607
{ymao1, ygu3, jchen5, grossman}@uic.edu

ABSTRACT

This paper presents FastPara, a C++ programming framework and associated runtime support for writing and running data-parallel applications in computer cluster environments. With FastPara, the user writes a declaration of the data to be exchanged between processes for a certain data-parallel processing algorithm. FastPara will handle all the process communications without any users' efforts. Complex data structures and variable-length arrays are well supported so that users can easily apply this framework to their data-parallel applications. FastPara's runtime provides built-in support for process management and fault tolerance for the parallel processing. FastPara can greatly simplify the development cycle. Based on the performance tests we did, FastPara demonstrated good communication efficiency and little process coordination overhead. We believe that FastPara is a very useful framework in developing data-parallel applications.

KEY WORDS
Parallel-programming, framework, data-parallel, cluster

1. Introduction

The use of clusters to run parallel applications is becoming an increasingly popular alternative to using specialized, typically expensive, parallel computing platforms because clusters provide a cost-efficient, reliable and scalable solution to high-performance computing problems [1].

Data-parallel [2] is a conceptually straight-forward parallel processing model. With this model, the solution of a problem on a large input data set is now obtained in three steps: split the input data into smaller portions; allocate a number of processes each working on an input data portion independently; and combine the results from each working process to produce the answer to the original problem. In practice, data-parallel can be applied to a large range of problems. For example, Fox [3] reported that data-parallel is the most common form of parallelism in scientific calculations.

Despite its simplicity in concept, writing and debugging a data-parallel program with an ad-hoc approach is time-consuming and the code is error-prone even if the data-parallel algorithm of the problem has been well understood by the developers [4]. Compared with traditional serial programs, extra efforts are required to handle communications and coordination between parallel processes.

In order to bridge the gap between the low programming efficiency and the high demands of parallel software, we designed and implemented FastPara, a declarative data-parallel programming framework. To use FastPara, developers make declarations to specify the input data to be split and the result data to be collected. FastPara will then generate the code that helps deal with data communications between processes. At runtime, FastPara will coordinate the running of those processes involved in a data-parallel program execution. With FastPara, all the work a developer needs to do is reduced to making data declarations, filling out the function of doing partial data processing and invoking the data-parallel processing by calling a produced function from the user program. FastPara is not a new programming language but a programming framework to support writing data-parallel applications in host languages (C++ at the moment). This feature makes it easier to integrate the parallel-computing related code with other parts of users' applications. Applying FastPara can greatly simplify the work for developing a data-parallel application. Users can freely download and install the source code distribution of FastPara from SoureForge.net [5].

The rest of this paper is organized as follows: Section 2 describes the basic usage of the programming framework from a developer's perspective. Section 3 explains advanced features. Section 4 evaluates the usability and runtime performance of the framework. In section 5 we present a real application case of FastPara. Section 6 is about related work and section 7 gives a short conclusion.

2. Framework Description

2.1 Runtime Process Control Paradigm

We decided to use the master/worker [6] process paradigm to support data-parallel processing. Mapping data-parallel processing to a master/worker process model is straight-forward: A master process is responsible for launching worker process, splitting data and collecting and merging results; a number of worker processes perform the desired processing on the data portion assigned to each of them. This is the basic structure of the process model and it will be augmented with more details in later sections.

2.2 Programming Model

In this section, we describe FastPara from the perspective of users, i.e., how to use it to write a data-parallel application. We begin with an overall outline of the process, followed by a concrete example.

We can follow 5 steps to write a data-parallel application in FastPara: 1) Design the data-parallel algorithm for the problem. 2) Write the declaration to describe the data to be exchanged and use the FastPara parser to generate code. 3) Finish the worker function implementation with appropriate processing logic. 4) Call the produced master function from the user's application. 5) Compile the master and worker process and link them to FastPara's libraries.

We now use a concrete example to illustrate the process. We intentionally chose a problem with a simple data-parallel solution as we want readers to stay focused on the framework itself rather than the algorithm. The example problem we are trying to solve is to count the occurrence of a given word in text. The text is broken into lines, each of which is stored in a character string. The solution to the problem can be obtained by counting the word occurrence in each character string and summing the counts.

First we design the data-parallel algorithm for this problem. We decided to split the input data, lines of text, into portions, count the word frequency in each portion and add up the numbers to get the final count. Since each line is represented by a string, we are splitting a number of strings into portions.

Second, we make the data declaration as in listing 1.

```
 1   parallel example.pwc
 2   {
 3       input
 4       {
 5           to_all:
 6               string word;
 7           to_split:
 8               string lines[];
 9       }
10       result (element)
11       {
12            int frequency;
13       }
14   };
```

Listing 1. *Parallel word counting example Declaration.*

Some remarks about the declaration in listing 1:

- All words in italic font are reserved keywords.
- Line 1 is the header of the parallel module. It uses an application name 'example' and a task name 'pwc'.
- In lines 5-6, we use ***to_all*** data (lines 5-6) to describe the data to be passed to every worker process. In most cases, it can be used to pass those global parameters for processing. In this example, since each worker searches for occurrences of the same given word, we put it in ***to_all*** data as a character string.
- In lines 7-8, we define the data to be split into portions for this data-parallel computation. We use a string array to represent the data we want to split for this problem.
- In the result part (lines 10-13), the keyword ***element*** means that for each data element (a character string in this example) from the to-be-split array, a result data is to be produced, which, in this example, is an integral number that holds the word's frequency count in the corresponding string.

Given the above declaration, the next step is to use FastPara's parser to do code generation. Some parts of the generated code are automatically used by FastPara runtime and users do not need to touch them at all. In this example, the users are only concerned about the worker member function and master invocation function.

The worker member function will be defined as a member of a generated worker class. At runtime, worker member function will be executed in each worker process to fulfill the processing on the data portion. The worker class name in this example is ***Cexample_pwc_Worker***, derived from the application and task names in the declaration. We give this member function's signature in listing 2.

```
int Cexample_pwc_Worker::worker_pwc(
        char * word,
        char ** lines,
        int  lines_len_,
        //result parameters follow
        int * frequency);
```

Listing 2. *Worker function declaration.*

Strings are mapped to null-terminated character sequences in C++. The first parameter ***word*** comes from the ***to_all*** part in listing 1. Since each worker will get a number of lines, we have the second parameter as an array of character pointers and the length of this array is given by the third parameter ***lines_len_***. Using a pointer plus a length-indicator integer is FastPara's approach to handling variable-length array data in C++. The last parameter ***frequency*** is for holding result data. In this example, we have declared that for each line, an integral count is to be produced, so when this worker function is called, parameter ***frequency*** will point to an integer buffer whose length is equal to the value of ***lines_len_***.

As we mentioned before, users should provide the implementation of this worker function to reflect the application's processing logic. For this example, we give an illustrative implementation in Appendix.

The master invocation function is for users to call from their application to start the parallel computation. The function declaration for this parallel word counting example is given in listing 3.

Like the worker member function, the input and output parameters for the master function are derived from the declaration made by users. So the first parameter is about the word to be searched for and the second and third parameters pass in the text to be searched in. The fourth parameter is an integer buffer for receiving results. On successful return, ***frequency[i]*** will hold the frequency count value for ***lines[i]***. The last two optional parameters can be used to specify a requirement on the minimum and optimal number of worker processes for the computation respectively. The return value of master function indicates if the processing is successful or not.

```
int example_pwc_master (
  char * word,
  char ** lines,
  int  lines_len_,
  //result parameters below
  int * frequency,
  int min_worker_num=1,
  int optimal_worker_num=AS_MANY_AS_POSSIBLE
   );
```

Listing 3. *Master function declaration*.

Unlike worker member functions, master functions' implementations are provided by FastPara and they serve as the calling interfaces to the parallel processing.

2.3 Program Execution

To run a FastPara application on a computer cluster, we need first have FastPara's agent processes running on those intended computer nodes. The responsibility of these agent processes is to launch local worker processes upon the requests from the master process. When starting up, each agent process will read a configuration file to get a list of worker processes that have been deployed on that machine. Assuming these agent processes have been started correctly, the following is what happens when the user runs the data-parallel application.

1. User program calls the master function, passing in the input data.
2. Master function reads a file named fp_master_init for communication parameters and agent list information. It then sends worker-process launching commands to found agents.
3. Agent processes will launch (through system call ***fork***) a number of local worker processes based on the commands from the master program.
4. Worker processes contact the master process and are each assigned a portion of the input data. In general, the master will try to split the ***to_split*** data array into equal-size portions so that the work is evenly distributed to worker processes. In section 3, we will see how users can customize the splitting logic.
5. Each worker process executes worker member function to do its local processing and produce the result.
6. Worker processes send their result to the master program, which assembles the result pieces.
7. On getting all the results from worker processes, master function returns to user program with the result data.

The above is a typical master process execution flow. There can be some variants and they will be mentioned when we describe the enhancing features. Figure 1 visualizes the execution flow.

2.4 Advanced Data Types

In the previous parallel word count example, we illustrated how to declare data of string and integer types. In real applications, the data exchanged between master and worker processes can be more complex. FastPara supports declaring data of complex structures.

There are three kinds of data types in FastPara: primitive types, array types, and composite types.

- Primitive types. There are five of them: integer, character, single floating-point number, double floating-point number, and string.
- Composite types. This basically corresponds to the ***struct*** data type in C++. The fields appearing in a composite type can be primitive types, array types or composite types. FastPara supports nested composite types so that users can define data of complex structures to unlimited levels.
- Array types. The type of array elements can be a primitive type or a composite type. Users can declare both fixed-length and variable-length arrays.

When the parser generates code for a declaration, it will map data types to appropriate C++ types which appear in the parameter lists of produced master function and worker member function. A corresponding C++ ***struct*** type is produced for each declared composite type. A variable-length array in a composite type is translated into a C++ pointer, plus an integer field giving the length of the array at runtime. Examples of composite types and variable-lengths and their mappings to C++ data types can be found in section 4.1.

Before we finish the discussion of data types, we want to emphasize that even though the syntax of data declarations may look like C++ and C++ is the only target language currently supported by FastPara, we still tried to make it language neutral so that in the future, this framework can be easily extended to other target programming languages.

2.5 Fault Tolerance Through Re-execution

In a data-parallel processing execution, some worker processes can fail for various reasons: the node can fail; the connection to the master process can be lost; the worker process code can also detect some internal error and decide to stop by returning from the worker function with an error code. The probability of having a failed worker process increases with the number of worker processes. In many circumstances, we can use the simple re-execution strategy to tackle the failed worker process problem. FastPara built this function into its runtime code so that this is easily available to the users without any coding efforts. When it detects a failure of any worker process, it will simply launch a replacing worker process and feed it with the input data of the failed worker process.

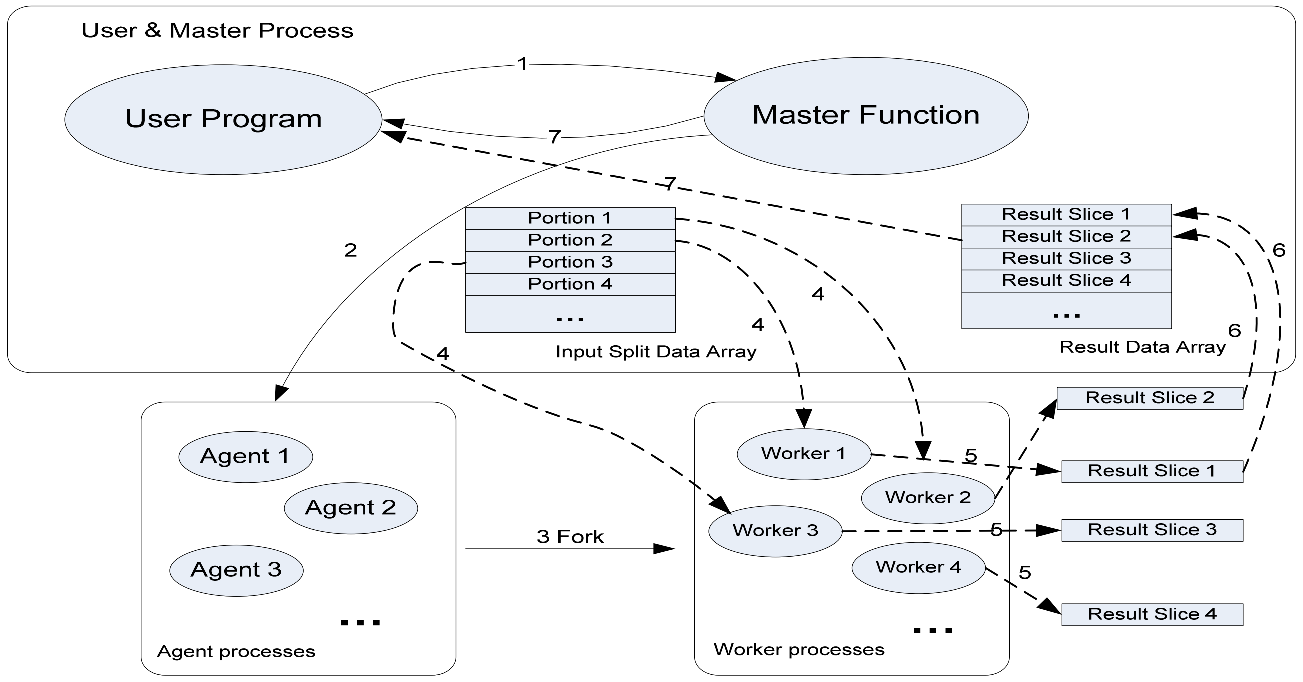


Figure 1. *Execution flow FastPara programs. Ellipses represent program modules. Rectangles represent data. Solid arrows mean control flows and dashed arrows mean data flows. Numbers in arrow labels indicate which execution step this flow occurs in, as described in section 2.3.*

When scheduling the replacing worker process, the master will try to schedule it to a different physical node from the failed process's node to minimize the probability of repeated failure.

3. Enhancing Features

The previous section introduces the basic functions of FastPara. We also designed and implemented a number of advanced features that enhance the usability of our framework.

3.1 User-customized Data Splitting

The splitting of input data (for to_split data) we have discussed so far always tries to assign an equal number of elements in the original array to each worker process. In FastPara, users can also completely define how the splitting should happen by setting a flag in the declaration. The master function produced in this case will take an integer array as an extra parameter. This array is called group size array. Each element in this array is a positive integer, indicating the length of a group in the original to_split array.

3.2 Incremental Result Processing

In some applications, the master process may want to process the results even when only some worker processes have returned their results. This is particularly true when the master needs to do so-called 'collective' operations on the results data. Citing the previous parallel word counting program as an example, we can actually have the master process initialize a total count value to 0 and whenever a worker process returns its result, we can add the counts from that worker process result to the total count. After the result data from all worker processes have been processed in this way, we get our answer. In most cases, this option is also better off in terms of runtime efficiency: it makes the master process do some useful processing rather than only wait for results.

This can be easily achieved with FastPara by making a small change to the data declaration and having the user implement a function of merging partial processing results.

3.3 Local Results Combining

In the parallel word counting example, each worker process actually does not need to return the word count for every line it received. A total count for all lines processed by the worker process is enough. So we can have the worker function to do local results combining and only return an integer value. This improves the efficiency due to the reduced size of data communicated between the master and worker processes.

3.4 Performance Profiling

For a parallel application, users care about the execution performance. To better help users analyze the performance and also get a complete view of the execution, FastPara tracks major events with their timestamps that occur during the run. It will log when the master function starts execution, when each worker process connects to the master and receives its input data and when each worker process sends result data to the master. It will also record any detected worker process failure and later re-executions of failed worker processes. All the profiling information is saved into a file that can be interpreted by a utility of FastPara.

3.5 Worker Process Initialization

For some applications, there is a need for worker processes to do initialization before executing the worker function and sometimes the parameters used by the initialization are specific to each worker process. A good example is that each worker needs to read files while the paths of these files vary on different machines. For this purpose, the user can define an ***init*** member function in the worker class and this ***init*** takes a string array as its input parameters. The user then puts correct values for these parameters in the worker-list files used by the agent processes on each machine. When the agent forks the worker process, it passes these parameter strings to the worker process so that it gets initialized appropriately.

4. Framework Evaluation

In this section, we focus our evaluation from two perspectives: programmability and runtime performance.

4.1 Programmability

Programmability is the most important advantage of our framework.

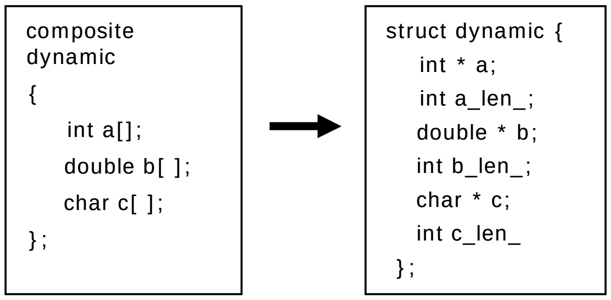
With FastPara, the complex task of writing and running data-parallel applications in cluster environments has been simplified to making data structure declarations, writing the partial data processing code and making calls to the master function, plus some trivial configuration file composing work (it mainly involves making a list of available agents for the master process) before execution. We also want to point out that the complexity of writing the FastPara declaration is pretty low since it basically involves composing the data types based on the application's data-parallel algorithm. The concise and simple interfaces of the master function and worker function free the user from writing the low-level code to handle communications. If we reexamine the steps of writing and running a data-parallel application with FastPara, we can see clearly that no communication details have been exposed to framework users. FastPara runtime also takes care of dynamic process management and fault-tolerance issues for worker processes automatically.

FastPara makes the user focus on how to split data logically and what result data should be produced, whose answers usually depend on the algorithm and the processing logic used by the application. If the user has to write the code from scratch to handle all the communication and process management issues, it is not only tedious, but also tends to result in error-prone code. Most data-parallel applications have a lot in common in their distributed processing model. FastPara therefore implements these shared patterns to spare users from reinventing the wheel for each individual application.

Another important practical enhancement FastPara made to improve programmability is its capability of handling complex data types in a declarative way. The types of data to be exchanged by worker and master processes are mainly decided by the application's processing logic. Anyone who has had experience with some message passing libraries will find that those libraries typically only provide direct support for communicating preliminary data types and users need to write non-trivial code to pass data with complex structures across processes. FastPara removes the necessity of writing this kind of code. The variable-length array feature we talked about in section 2.4 is a great convenience for users. To make this point clear, we draw a comparison to MPI [7] with an example.

Assume in a certain data-parallel application, each element of the input array is of the composite type ***dynamic*** illustrated in the left box of figure 2.

The ***struct*** definition in the right box of the figure 2 is the outcome from FastPara's code generator. In this example, a composite type dynamic contains three variable-length arrays: ***a***, ***b*** and ***c.*** In the produced C++ ***struct***, fields ***a_len_***, ***b_len_*** and ***c_len_*** are inserted by FastPara's parser automatically to give the arrays' runtime sizes.



```
composite
dynamic
{
    int a[];
    double b[ ];
    char c[ ];
};
```

```
struct dynamic {
    int * a;
    int a_len_;
    double * b;
    int b_len_;
    char * c;
    int c_len_
};
```

Figure 2. *Variable-length arrays in FastPara. The left box gives the declaration of a composite type of three variable-length fields. The right box presents the generated C++ struct from that declaration.*

Given the above data types, the master process may have the following input array to distribute.

```
struct dynamic   data[100];
...
xxx_master (..., data, ...) ;
                    //call the master function
                    //with 'data' as input array
```

In FastPara, it is quite all right if data[***i***].a_size != data[***j***].a_size (***i***!=***j***). The runtime detects and handles these variable-length arrays appropriately without any manual coding.

However, if we want to distribute the array ***data*** within MPI, it can become much trickier. First, MPI needs the explicit code to generate a so-called ***datatype*** object for any complex data structure. Second, the length of any array field in a created MPI ***datatype*** object must be a fixed value. So if two elements in ***data*** array have different ***a_len_*** (or ***b_len_*** or ***c_len_***) values, the user must then create two different ***datatype*** objects. For our ***dynamic*** example, the user may end up creating one ***datatype*** object for each element in the array. What makes it even worse is that for any MPI communication involving a certain ***datatype*** object, both the sending and receiving process must create and commit this ***datatype*** object. For that purpose, usually the user may have to write the code to make the sending process and the receiving process exchange this array size information. We will revisit this ***dynamic*** data structure when investigating the performance of transmitting variable-length array data later on.

To summarize, FastPara provides great programming support for data-parallel applications by taking care of many low-level details that used to require tedious work from developers.

4.2 Performance

To correctly evaluate the runtime performance of FastPara, we should focus on the runtime process communications and management efficiency rather than the efficiency of the parallel algorithms themselves. The basic communication patterns involved in FastPara execution are scattering input data and receiving partial result data. We designed the following two tests to evaluate FastPara's runtime performance. In the first test, we used a master process to split an integer array into portions and distribute them to a number of slave processes and had the master process collect all those portions from slave processes. In the second test, we split and distributed an array (say ***d_array***) whose elements are of the composite type ***dynamic*** described in section 5.1 and had slave processes each return a single integer value to the master process. We made the size of ***a***, ***b*** and ***c*** fields of ith element ***d_array***[i] in the array equal to i+1. Using variable-length arrays in type ***dynamic*** means FastPara runtime needs to transfer non-memory-contiguous data since array pointers may point to non-adjacent memory blocks. In both tests, we tested arrays with different sizes so that we can have a comprehensive understanding about the performance under different data communication volumes.

As a comparison, we made logic-equivalent implementations for the above two tests using MPI. We tried two popular MPI implementation versions: LAM/MPI[8] with version 7.1.1 and MPICH2[9] with version 1.0.3. We run the programs in the same Linux cluster of 10 nodes. Each node runs Debian Linux with kernel version 2.6.8 on Intel Pentium IV at 2.8 GHz with 1G memory. The networks connecting these nodes are Ethernet at 100 Mbps.

Table 1 and table 2 give the raw program execution times in seconds with three implementations in two tests respectively. In test 1, when the array size is n, the size the data involved in transmission is about n*4*2 (It is multiplied by 2 since the data are first distributed and then collected). In test 2, when the array size is n, the size of distributed data is n*(n+1)*13/2 (since the size of ith element in the array is 13*(i+1)). So the largest size we used for testing is 800M bytes and 650M bytes respectively in the two experiments. They are more than the data volume size to be transferred in most data-parallel applications. Figure 3 and figure 4 use graphs to compare performances of different implementations. In the two graphs, the execution time of LAM/MPI is normalized to 1.

Array Size	10^2	10^3	10^4	10^5	10^6	10^7	10^8
FastPara	0.017	0.020	0.098	0.108	0.653	5.58	56.4
LAM/MPI	0.628	0.632	0.633	0.699	1.240	6.01	59.5
MPICH	0.254	0.310	0.321	0.358	0.930	5.73	58.0

Table 1. *Time performance data for tests of distributing integer arrays on a 10-node cluster. The first row gives the size of the integer array. The second, third and fourth rows give the execution times in seconds for FastPara, LAM/MPI and MPICH implementations respectively.*

Array Size	1000	2500	4000	5500	7000	8500	10000
FastPara	0.511	2.90	7.20	13.4	21.2	33.5	44.7
LAM/MPI	1.19	4.14	9.52	17.8	28.5	42.3	58.5
MPICH	0.918	3.74	9.08	16.9	27.1	39.8	54.8

Table 2. *Time performance data for tests of distributing arrays whose elements contain variable-length arrays. The first row gives the size of the arrays to be distributed. The second, third and fourth rows give the execution times in seconds for FastPara, LAM/MPI and MPICH implementations respectively.*

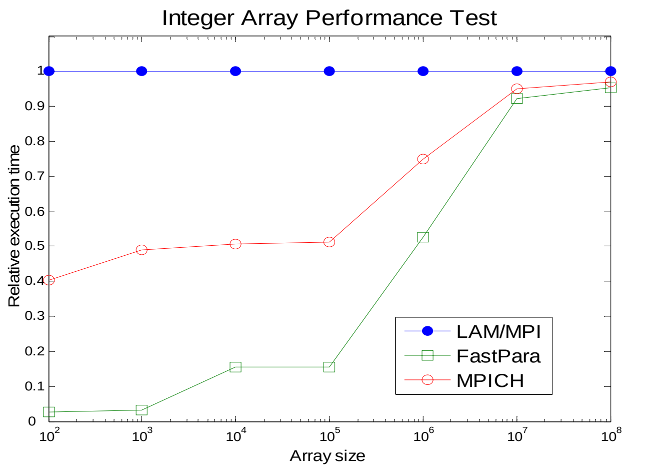


Figure 3. *Execution time comparison for splitting integer arrays. Plots the relative execution time for the performance test described in this section. The execution times of LAM/MPI are normalized to 1.*

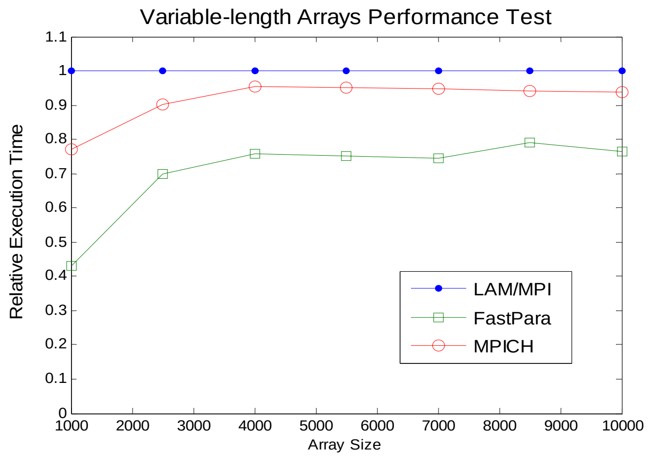


Figure 4. *Execution time comparison for handling variable-length array data. Plots the relative execution time for the performance test described in this section. The execution times of LAM/MPI are normalized to 1.*

One interesting discovery from the performance comparison is that FastPara exhibits much better overall application time performance than the two MPI implementations when the data size is relatively small or medium. The major loss in the two MPI implementations comes from the time on MPI process setting up. A rational reason for this is that in FastPara, the relationship between processes is one to many: each worker process only needs to set up the communication channel with the master process. In MPI, however, it needs to set up a many to many communication graph: MPI assumes each process may need to exchange data with any other process. Obviously, FastPara provides better support in this aspect for data-parallel applications since it takes advantage of the communication pattern. When the data size is large enough, the impact of this performance advantage fades away and FastPara shows performance close to the two MPI implementations.

5. An Application Case

As a real world case, we successfully applied FastPara to an open-source project for developing a protein identification program SPARK [10]. It takes the experimental data, which are mass spectrum values, of an unknown protein sample and tries to identify the protein. The program computes the theoretical tandem mass spectrum data for each entry in a given protein database and then compares the theoretical data to the experimental data. The entries with top matches will be returned as the search result. The search algorithm we use scans the database twice. The first scan serves as a filtering step for the second scan. In the first scan, a brief comparison is made so that a portion of the whole protein database is singled out. In the second scan, the program makes a more deliberate search in the proteins found in the first scan to produce the final result.

Due to the large size of the protein database and high computational complexity, we made a parallel version using FastPara. The implementation work turned out to be rather smooth and required little debugging efforts. Our parallel SPARK program runs on a 4-node apple cluster with the following hardware configuration. Each node contains dual Power G5 processors at 2 GHz with 4G memory. The network connection between nodes is 1 Gb/second. Table 3 compares the execution times between the serial and parallel version of SPARK with 5 different testing data files.

Test	**1**	**2**	**3**	**4**	**5**
Serial SPARK	662	769	821	453	377
Parallel SPARK	91	101	113	82	51

Table 3. *Search time comparison between serial SPARK and FastPara-based parallel SPARK implementation. Parallel SPARK implementation runs on a 4-node (8 processors) Apple cluster. All times are in seconds.*

6. Related Work

Broadly speaking, there are three types of work that inspired and/or influenced the design and implementation of FastPara.

The first type of work is general purpose message passing libraries. MPI [7] and PVM [11] are two examples. The basic function of these libraries is to enable exchange of data between processes. They are preferred by many developers for doing process communications due to their high runtime efficiency and wide platform support. Compared to FastPara, these libraries are much less programmable when writing data-parallel applications. They require much more effort from developers to handle data with complex structures. They also provide very weak support for the runtime process control and fault-tolerance desired by most data-parallel applications.

As we mentioned, FastPara provides programming support in a framework approach. This makes it quite distinct from many parallel programming languages like UPC [12],mpC [13],C** [14],HPF [15] and NESL [16].These parallel programming languages usually introduce new language syntaxes to support parallel/distributed computing. This not only tends to make these languages lack user acceptance [17], but also adds to the complexity of integrating parallel and non-parallel parts of the applications. In FastPara, the only part that exposes users to parallel programming details is the data distribution declarations. For other parts (writing the worker function and calling the produced master function), users design and write the code in the traditional serial program context. So FastPara provides better abstraction in writing data-parallel applications.

The third work type is some parallel programming frameworks which, like FastPara, try to encapsulate the recurring patterns in parallel applications to provide users with more abstract and concise programming interfaces. They are discussed in the following several paragraphs.

MapReduce [18] is the programming framework used in Google to do distributed data processing on clusters. Like FastPara, its runtime handles process coordination (including fault-tolerance for worker processes) and communication automatically and the user is only concerned about data processing logic on each node. Unlike FastPara, MapReduce mainly uses disk files as the input sources and result destinations and the MapReduce framework extracts data from these files as strings. Users need to write their own conversion functions when the application needs to handle non-string data.

MW [6] is a software framework to help build a master-worker style application. It handles the creation and management of master and worker processes at runtime automatically. MW requires users to write code to split data into pieces and the data exchanged between processes must be packed/unpacked with user-defined code. MW's interface to the user program is much more complicated than FastPara: users need to implement 10 functions defined in 3 abstract classes for a single application.

Trellis-SDP [19] is another data-parallel programming interface. Like MapReduce, it has worker processes to read input data from the file system. But the result of worker processes will be transferred back to master process through binary streams. Again, it does not directly support user-defined data. And its runtime does not have fault-tolerance for worker processes.

7. Conclusions

The main contribution of FastPara comes from its declarative approach to data-parallel programming. It hides a lot of low-level coding details from the developers and its built-in support for runtime process management and fault-tolerance helps achieve the desired runtime behaviors for data-parallel applications which otherwise would have required a lot of manual coding work by developers. Applying FastPara can greatly improve the programming efficiency in developing data-parallel applications.

Acknowledgment

This work was supported, in part, by funding from The Searle Funds at The Chicago Community Trust.

References

[1] R. Buyya, *High Performance Cluster Computing: Architectures and Systems* (1st edition Prentice Hall, PTR,1999)

[2] A. Grama, G. Karypis, V. Kumar, and A. Gupta,*An Introduction to Parallel Computing: Design and Analysis of Algorithms* (2nd Edition, Addison Wesley, 2003)

[3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors* (Englewood Cliffs,Prentice-Hall, NJ, 1988)

[4] R. Buyya, *High Performance Cluster Computing: Programming and Applications* (1st edition Prentice Hall PTR,. 1999)

[5] FastPara Information and Download Homepage, http://sourceforge.net/projects/fastpara .

[6] J. P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder, An enabling framework for masterworker applications on the computational grid, Tech. Report, University of Wisconsin –Madison, March, 2000

[7] Message Passing Interface Forum: MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[8] G. Burns, R. Daoud, and J. Vaigl, LAM: An Open Cluster Environment for MPI, *Proceedings of Supercomputing Symposium*, pp. 379--386, 1994.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6), pp. 789-828, 1996.

[10] R. Grossman,J. Chen, Y. Mao, D. Hanley, and A. Schilling, The SPARK Protein Identification Algorithm, submitted to a special issue of *ALGORITHMICA on Algorithmic Methodologies for processing Protein Structures, Sequences and Networks*.

[11] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, 2(4), pp 315--339, December, 1990.

[12] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and Language Specification.* CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[13] A. Lastovetsky, mpC - a Multi-Paradigm Programming Language for Massively Parallel Computers, *ACM SIGPLAN Notices*, 31(2): pp. 13-20.

[14] J. R. Larus, B. Richards, and G. Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language, *UW Technical Report 1126*, Computer Sciences Department, University of Wisconsin--Madison, Madison, WI, November 1992.

[15] D. Loveman, High Performance Fortran, *IEEE Parallel & Distributed Technology*, pages 25-42, February 1993.

[16] G.E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zegha, Implementation of a Portable Nested Data-Parallel Language, *4th Symp. Principles & Practice of Parallel Programming (PPoPP)*, pp. 102-111, May 1993.

[17] B. Freisleben, T. Kielmann. Approaches to Support Parallel Programming on Workstation Clusters: A Survey. Informatik-Bericht Nr. 95-01, University of Siegen, Dept. of Electrical Engineering and Computer Science, Siegen, Germany. 1995.

[18] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of OSDI 2004*, pp. 137-150.

[19] M. Ding, P. Lu: Trellis-SDP: A Simple Data-Parallel Programming Interface,*ICPP Workshops 2004*, pp.498-505.

Appendix

An example implementation of worker function for parallel word counting in section 2.2

```
int   Cexample_pwc_Worker::worker_pwc(
            char * word,
            char ** lines,
            int  lines_len_,
            //result parameters follow
            int * frequency)
{
   int count; char *s;
   int word_len=strlen(word);
   for (int i=0;i<lines_len_;i++)
   {
        count=0;   s=*lines++;
        while (s=strstr(s,word))
        {
             count++;
             s+=word_len;
        }
        frequency[i]=count;
   }
   return WORKER_SUCCESS;
}
```