A Generic Parallel Map:  
Extending the List Map Higher Order Function to Arbitrary Data Structures

Mohammad M. Hamdan*

Received on July 12, 2005 Accepted for publication on Feb. 28, 2006

Abstract

The standard map higher order function is commonly used by the parallel functional programming community to apply a function to the elements of a list data structure in parallel. In this work, a generic parallel map was developed. The design and implementation of the generic parallel map was based on the integration of a sequential generic functional programming language, a data distribution algorithm, a parallel master slave template and generic sends. The new generic parallel map can take a function and apply it on any data type. Results are shown for parallelizing a ray tracer using the generic parallel map. Speedup results are good.

Keywords: functional programming, parallel processing, generic programming, data distribution, templates, ray tracing, genetic algorithms.

Introduction

There is a big need within the scientific computing community for programming methodologies that facilitate the development of computer application that can run sequentially and possibly in parallel. These methodologies are required as programmers need to focus on the problem being solved rather than looking at how to speed their program. Therefore, the idea of skeletons or templates or even patterns [1,2] was welcomed by the scientific computing community because of their implicit parallelism.

However, the work on templates has primarily focused on parallelizing lists. Most of the problems had to be converted from their original data structure representation into a list. For example, a vector can be represented using a list and a tree can be represented using a list of lists. This conversion process is expensive as it requires the programmer to think of the problem in a different format and the algorithm had to be modified to work on the new data structure.

Previous work on generic functions provided either sequential functions such as PolyP [3] and Generic Haskell [4] that can work on any data type or parallel functions...
using an explicit approach such as Janus [5] and extending C++ standard template library [6]. These approaches are compared to the work in this paper in Section 4.

Using the power of generic programming [7,8] which is available in Fish [9] and the basic ideas in parallel algorithmic skeletons enabled us to implement a generic parallel map (called gpmap). The gpmap can be used with complex data structures and its underlying parallel implementation will do the work in parallel.

The rest of the paper is organized as follows. Section 2 describes how the generic parallel map was engineered. The results are shown in Section 3. In Section 4, comparison to related work is outlined and finally Section 5 concludes and outlines future work.

**Engineering a Generic Parallel Map**

The generic parallel map was a result of the integration of a host language (Fish), a generic sequential map, a static data distribution algorithm, a static master slave template and generic sends. The following subsections will explain each part in detail.

**The Host Language**

A generic function is a function which is able to handle any data structure containing any kind of numerical data such as integers, floats, Boolean, lists … etc. It is polymorphic in the choice of the structure used to hold the data. To explain the idea of generic programming [3, 5], take a look at the following example:

```
equal 1 2
equal 3.4 6.7
equal true false
equal [1,2,3] [1,2,3]
```

As we can see, arguments for the function equal can be of different types. There is one implementation for the function and it can take many different data types as arguments. This is similar to the idea of overloading functions and templates in C++ [10]. However, we are interested in a functional language rather than an imperative or an object-oriented language due to the advantages of using functional languages such as the declarative style which matches our goal of implicit parallelism.

The host language for this work is a well known functional programming language called Fish [9]. Fish stands for Functional = Imperative + Shape and it is designed to be an array programming language. Its main features are the expressive power of functional languages and the efficient execution of imperative languages. The power in Fish comes from static shape analysis [11], which is done at compile-time in order to reduce functional programs to simple imperative forms. Moreover, it supports generic programming using the idea of constructor calculus [12] which is based on generic pattern matching.
Fish was chosen for the gpmap due to the availability of the source code for the compiler and its support for generic programming. The Fish code shown below, illustrates how to define a binary tree with two data types: real for leaf nodes and integer for root and intermediate nodes. Its graphical visualization is shown in Fig. 1.

datatype btree (a,b) = leaf a | node b : btree(a,b) : btree (a,b)

let Tree = node 4 (node 6 (leaf 3.2)
    (node 10 (leaf 4.8) (leaf 1.3)))

The tree can hold mixed data types such as integers and reals by using the constructor calculus [12]. The constructor calculus allows generic pattern matching to branch on any constructor of any type.

Fig.1: The tree data structure with mixed data types

A Generic Sequential Map

In Fish there is a built in generic sequential map that can apply a function or more to a data structure that contains more than one data type. To illustrate how the generic map works, an example is shown below. In the code we define two functions: the first function fun1 takes an integer and returns an integer while the second function fun2 takes a real and returns a real. The generic map can take a tuple of these two functions and apply them to the values stored in the tree. The resulting tree is shown in Fig. 2.

let fun1 x = x + 3
let fun2 y = y +. 4.5
map (fun1,fun2) Tree
Tree = node 7 (node 9 (leaf 7.7)
    (node 8 (leaf 7.0) (leaf 8.2)))
    (node 13 (leaf 9.3) (leaf 5.8))
Fig 2: Applying a tuple of two functions to the mixed data type tree using the sequential generic map.

Data Distribution

The basic idea of data distribution [13] is to take a data structure and distribute it across a given number of processors. In case of an array or a list this is an easy task where we divide the number of elements over the number of processors. Each processor will get its piece of the array accordingly. The advantages of data distribution are load balancing and improve data locality. Nonetheless, few processors might get less or more data than the others and this might cause load imbalance. However, the problem with data distribution is how to distribute complex data structures such as trees. There are two approaches: the first one is to flatten the data structure to bytes which is quite expensive and looses locality of data and might have extra overhead due to load imbalance and the second one is generic which is not expensive and preserves locality of data but difficult to implement.

A compile-time algorithm was developed that distributes a tree across a given number of processors. We illustrate in Fig. 3 a distribution for the tree data structure that was defined in section 3.1 and a flow chart for the algorithm is shown in Fig. 4.

Fig. 3: Distributing the tree data structure across two processors.
The result of the algorithm is a list of lists of pieces as shown in Fig. 5. Each sub-list will be assigned to a processor and hopefully amount of work is balanced across all processors.
Fig. 5: Generating a list of lists of unprocessed pieces.

The result of applying the gpmap on the tree that was defined in Section 2.1 would be as shown in Fig. 6. The final step is to reconstruct the tree from the processed list of lists of processed pieces.

Fig. 6: The processed list of lists of piece

The Static Master Slave Template

Template-oriented parallel programming is a well methodology for parallel programming. It has two components: a high-level interface for templates that the programmer can use in his program in order to parallelize it and a low-level implementation that will be used to parallelize the high-level interface. The user is not responsible for any coordination, messages, load balancing or any other parallel programming issues. All these are taken care of using the pre-implemented template. The programmer is only responsible for choosing the suitable template and instantiate it with the needed function and data. The choice can be from a variety of templates such as the master slave, pipeline, divide and conquer and geometric parallelism. These templates correspond to the common algorithms that are normally used in developing parallel applications. To illustrate the methodology lets take the map Higher-Order Function (HOF) as an example. The user may in his program apply a function to the elements of a list. This can be easily done using the map HOF as follows:

```haskell
fun square x = x * x;
map square [1, 2, 3, 4, 5] =>
[square 1, square 2, square 3, square 4, square 5] =>
[1, 4, 9, 16, 25]
```
We may note that the square function can be applied to each element in the list simultaneously as each element is independent from the other one. This type of implicit parallelism can be easily exploited by the compiler using a master slave template. The parallelizing compiler will replace the map function with the master slave template in the generated object code. Once the program is executed the square function will be applied to the elements of the list in parallel.

We have implemented modified version of the static master slave template for the generic parallel map. The master sends a list of pieces to each worker. Each slave will receive its list of pieces, process it and send the result back to the farmer. Note that the farmer after the send process will do work as well. Once the master finishes its computation then it is ready to receive the result from the workers. The pseudo-code for the master and slave template are shown below:

**The Master Alg.**
1) Send the nth list of pieces to the nth worker
2) Process the 1st list of pieces
3) Receive the kth list of pieces from the kth worker
4) Build a list of lists of processed pieces

**The Worker Alg.**
1) Receive the nth list of pieces from Master
2) Process the nth list of pieces
3) Send the list of processed pieces to Master

**Generic Sends**

The gpmap requires generic sends. Therefore, it was required to add functions to be used by the master slave template that can send and receive complex data structures. Since any Fish data structure can be translated using the shape information into one or up to four basic arrays (int, float, char, Boolean). Simply, to send arbitrary data structures: either send the four arrays using separate messages or combine the four arrays into one array and transmit the data structure using one send message. The receive process is similar. It important to note that there is no need to flatten or unflatten a data structure at run-time.

Using the message passing interface library (MPI) [14] it was possible to combine the four arrays and send them as one big message. This was achieved by defining a new data structure composed from the four arrays using MPI’s built-in datatypes. The new data structure is then send using a standard send routine.

**Results**

To test the generic parallel map, we have implemented a ray tracer in Fish. The ray tracer is a well known example that has been used by the parallel functional
programming community as an application to test their methodologies and techniques. Therefore, the program was converted from ML [15] to Fish. The original program was taken from Impala suite [16] for parallel benchmark programs.

The basic idea of the program is to generate a 2D image from a 3D scene which consists of spheres. In the resulting 2D image each pixel in the grid will be colored according to traced ray for that pixel. During the tracing of a ray, all intersections of this ray with the given objects are computed. The ray is reflected once an intersection is found and the color of the intersection point is computed based on the strength of the ray. The program was tested at Heriot-Watt University's Beowulf cluster which is based on Intel Celeron 533MHz, 128KB cache, 128 of DRAM and 5.7GB of IDE disk. The workstations are connected through a 100 Mb/s fast Ethernet switch.

Fig 7 shows the execution time and speedup for the ray tracing of an image (512 x 512) consisting of 9 Spheres and 2 light sources. Speedup are results are good and better than those presented in [17] up to 6 processors. However, there is drop in performance when 8 processors where used. This might be caused by the increase in size of the resulting object code and possibly less utilization of the cache. It is important to note that this is caused by the FISh compiler as it generated bigger code size as we increase the number of processors. It is possible to solve this issue by adding run-time shape analysis and not relying completely on compile time analysis. The resulting ray traced spheres are shown in Fig. 8.

![Ray Tracer Execution Time](image1)

![Ray Tracer Speedup](image2)

**Fig. 7:** The execution time and speedup for the ray tracer program.
Fig. 8: The result of applying the ray tracer on a scene consisting of 9 spheres and 2 light sources.

Comparison with other Models

Table 1 presents a comparison between gpmap and other models. It is important to note that gpmap differs from most of these models as it is based on a functional language and it has the advantage of the implicit parallelism that is available in HOFs. The other implementations are semi-explicit and rely on the programmer to parallelize the sequential application. Therefore, it is expected that gpmap will be preferred by programmers as it encapsulates implicit parallelism and the user does not have to worry about communication, coordination and any parallel processing issues.

Table 1: Comparison between gpmap and other models.

<table>
<thead>
<tr>
<th>Models</th>
<th>Functional or Imperative</th>
<th>List Data Structure</th>
<th>Generic Data Structures</th>
<th>Supports Parallelism</th>
<th>Parallelism Type</th>
<th>Run-time Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpmap</td>
<td>Functional</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Implicit</td>
<td>Skeleton-library</td>
</tr>
<tr>
<td>Janus</td>
<td>Imperative</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PolyP</td>
<td>Imperative</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Generic Haskell</td>
<td>Functional</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PMLS [18]</td>
<td>Functional</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Implicit</td>
<td>Skeleton-library</td>
</tr>
<tr>
<td>GpH [19]</td>
<td>Functional</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Semi-Implicit</td>
<td>Graph-Reduction</td>
</tr>
<tr>
<td>Eden [20]</td>
<td>Functional</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Semi-Implicit</td>
<td>Graph-Reduction</td>
</tr>
</tbody>
</table>
5 Conclusions and Future Work

It was possible to design and implement a generic parallel map that can apply a function to arbitrary data structures. It was engineered from different components: a host language, a generic programming, master slave template, data distribution and generic sends. It was used on a decent example (the ray tracing of spheres) which consisted of a slightly complex data structure due to the lack of description of applications using generic programming languages [21]. It is planned to use the generic parallel map for further examples. Also it is possible to think of adding cost models with real hardware parameters to the generic distribution.

It is important to note that a good application for the gpmap would be in the parallelization of genetic algorithms Gas [22]. To explain, in a GA there are different types of representations such as binary, float, arithmetic, trees … etc according to the encoded problem. The GA operators such as the crossover operator is nearly the same operation regardless of the representation method. Therefore, a generic crossover could be implemented that works on different datatypes (representations) and used a function argument for gpmap to applied in parallel on the population strings.

Acknowledgment

The author would like to thank Dr. C. B. Jay and Dr. Q. T. Nguyen from University of Technology, Sydney, Australia for access to the Fish compiler and the useful discussions during the development of this work. Also, the author would like to thank Dr. Greg Michaelson from Heriot-Watt University for giving the author access to Heriot-Watt's Beowulf cluster.
References


Hamdan


134