ABSTRACT

This paper introduces a new methodology for parallel programming. The methodology is an extension to the original methodology proposed by Darlington et al. in 1992, which is based on the concept of skeletons. The main aspect of the extension is precisely a set of alternative implementations for such skeletons. In particular, we designed, implemented, and studied the performance of alternative implementations for the Pipe, Farm, and Divide and Conquer skeletons. These studies show that it is indeed possible to improve the performance of the original implementation in specific scenarios by taking into account problem parameters such as size of the data structures, type of communication between processors, bottleneck processes, granularity, number of available processors, complexity of the functions involved, etc.

1.- INTRODUCTION

One of the main problems with parallel machines is the cost and complexity of the software development process. Programming parallel machines is more difficult than programming sequential machines in at least two fundamental ways: portability and predictability of performance[DAR92], due to the diversity of parallel machine models.

A programming methodology for parallel machines, which allows portability both of programs and their performance across the whole range of architectures was presented by Darlington et al.[DAR92]. This methodology is based in a set of high-order parallel forms called ‘skeletons’.

The objective of this paper is to propose alternative implementations for some of these ‘skeletons’, and to support the programmer when choosing among them depending on the current parallel problem. As a result, we present an extension of the original methodology.

In the von Neumann model of computation, resource allocation, for instance, is done by the compiler without any performance cost. Therefore, the programmer can forget about low level issues and still predict very accurately the program performance. On the other hand, in parallel programs resource

1Acknowledgements: This project was partially funded by Pontificia Universidad Católica de Chile, under Grant DIUC 93/05E.
allocation is one of the key points in the performance, so the programmer needs to have control over it. With regard to portability, again the von Neumann paradigm offers transparency—at the language level—to move programs from one machine to another, without any unpredictable performance change. However, in parallel systems because the resource allocation is done by the programmer, the programs are machine dependent, and there is no portability at all. Even when some programs can be compiled in different machines the performance will be unpredictable anyway.

The key of the methodology presented by Darlington et al. is the abandonment of the search for portability at the language level in favour of a structured decision-making process based on the use of high-level program forms, source-level program transformation and performance modelling. Put together, these factors allow the programmer to specify applications at a very high level and systematically to target and re-target the specification at particular machines without resorting to low-level machine-dependent programming. At the same time the programmer has enough control to achieve a high level of efficiency and to reliably predict the performance of any particular instantiation of the program.

This paper is organised as follows: section 2 presents the conceptual framework of the methodology; sections 3 to 5 analyse the original implementation of some skeletons and propose new implementations for each of them; section 6 compares the results obtained by executing parallel programs using different implementations for the skeletons; finally, section 7 gives some concluding remarks.

2.- CONCEPTUAL FRAMEWORK

Methodology

Darlington’s programming methodology for parallel machines [DAR92] is based on a set of ‘prepackaged’ units, named skeletons, that represent natural structures of parallel algorithms. This means that in general it is possible to adapt a potential parallel program to one of these skeletons. On the other hand, each of this skeletons has associated a performance model with every known parallel machine. The methodology can be reviewed in three steps:

1) to write the specification of the application in a functional programming style, using the predefined skeletons as high order functions.

2) to find the more suitable skeletons, in terms of performance, for the parallel machine to be used.

3) to use program transformation techniques to convert, algorithmically, the program of point (1) into a new program that uses the skeletons defined in point (2). In other words, to transform machine-independent general functional programs into equivalent forms expressed using the appropriate machine-specific skeletons.

The key issue of this methodology, from the point of view of the user, is that the programmer specifies programs at a very high level and target the specification to a particular machine. The programmer can predict the performance of the program on that machine. The programmer follows a structured decision-making process using high level programs forms, program transformation and performance modelling.
Skeletons

The skeletons can be described as the parallel unit of this programming methodology. The skeletons have associated a performance model (which is machine dependent) and they have the responsibility of all the resource allocation. In other words, the skeletons tackle the two main problems mentioned above: portability and predictability of performance.

An initial set of 5 skeletons has been implemented to define the most common patterns of parallel algorithms.

Pipe

The pipe skeleton defines a simple linear process parallelism. A list of functions are composed together so that elements can be streamed through them. Parallelism is achieved by allocating each function to a different processor. An example application for the Pipe skeleton is the function compile, which defines the compilation route for a high-level language.

compile = PIPE [writefile, typeCheck, parse, lex, readfile]

Farm

The farm skeleton captures the simplest form of data parallelism. A function is applied to each of a list of 'jobs'. The function also takes an environment, which represents data which is common to all of the jobs. Parallelism is achieved by utilising multiple processors to evaluate the jobs. An example application for the Farm skeleton is the function exposedFaces, which determines the faces of a convex 3-dimensional body are visible from one point of the coordinate system. Each face is checked in parallel, and the coordinates of the point form the shared environment.

exposedFaces = FARM [checkIfVisible, point]

Divide and Conquer

The divide and conquer skeleton follows the idea of splitting a task in several sub-tasks, solving the sub-tasks independently, and combining the results. Parallelism is achieved by allocating each sub-task in a different processor. An example application for the Divide and Conquer skeleton is the function mergesort, which works recursively splitting a list in trivially sorted sub-lists, and merging them, obtaining the sorted permutation of the original list.

Reduce and Map over Pairs (RaMP)

The RaMP skeleton describes those systems where each object in the system can potentially interact with any other object. Each individual interaction is calculated and the results are combined to produce a result for each object. An example application for the RaMP skeleton is a system of heavenly bodies interacting via gravitation. The force between each pair of bodies is calculated and these are summed to determine the total force acting on each body and its new position and velocity.
Dynamic Message Passing Architecture (DMPA)

The DMPA skeleton describes systems where any process can interact directly with any other process via message-passing, the actual connections being determined using run-time data. Parallelism arises from evaluating the processes on different processors. An example application of the DMPA skeleton is a database distributed over a network of processors. Each node has to be capable of handling requests for the whole database.

Extended Methodology

The methodology presented by Darlington et al. [DAR92] gives the programmer the possibility to use a unique implementation for each skeleton. But, as we will show in this paper, only one implementation for each skeleton is not efficient enough for every possible case. We have extended the original methodology, so that after selecting a particular skeleton, the programmer can select the skeleton implementation more suitable to his problem. In order to make this decision the programmer is given a set of alternative implementations, each one with the associated performance model. The programmer, using the parameters relative to his particular problem, evaluates the different performance models and selects the proper alternative.

The alternative implementations must be analysed from the point of view of the different kinds of applications using the same skeleton, and considering factors such as: size of the data structures, type of communication between processors, bottleneck processes, granularity, number of available processors, size of the problem, etc. These are all standards factors in the design and implementation of parallel programs.

At the end, for each skeleton the programmer will have a number of options to chose from at compile time. Each option represents a different implementation of the skeleton. Each implementation represents a certain category of problems. For instance, suppose we have a problem to be solved using the pipe skeleton. Before compiling the application it is necessary to study the data structure to be transmitted among the stages, the number of stages of the pipe, the heaviest functions to be applied, etc. Then according to this analysis, we can decide the better implementation of the pipe to be used. This decision will be supported by an analytical model [GAH91] that will predict the performance of the program given its characteristics.

3.- IMPLEMENTATIONS FOR THE PIPE SKELETON

Pipe Original Implementation

The pipe skeleton receives as input two parameters: a list of functions which defines the stages of the pipeline and a list of values which will be processed by it. The output is a list of values processed by the pipe.

The Pipe execution can be divided in three sequential processes: Set-up, Process and Termination.
Set Up

The set-up performs mainly the search for free processors and the allocation of the functions to them. The following are the main steps for the set-up phase:

(1) Find out the slaves that are free at this point. An AREYOUFREE message is sent to every processor and waits for the replies, building a list with the free processors.

(2) An ALLOCATE message is sent to every free processor in the list and waits for the replies, building a list with the allocated processors.

(3) Using the number of allocated processors (allProc) and the number of functions in the pipe (nFunc), it is calculated the number of stages in the Pipe (pStages):

\[ pStages = \min(\text{allProc}, \text{nFunc}) \]

(4) Finally, the functions are allocated to the slaves. The master sends a message to each slave indicating the next slave in the pipe and the functions to be performed.

Process

The process feeds the pipe with the data. For each element in the list of data (second argument of the pipe), the master sends a message to the first slave in the sequence including the value to be processed and a tag indicating if this is the last value or not.

The processed values received from the last slave are appended to the result list. On the other hand, each slave receives the data, processes it with the corresponding pipe functions contained in it and sends the subprocessed data to the next slave (or the master in the case of the last slave in the sequence).

Termination

Finally, the termination stage frees all the slaves by sending a deallocating message to each of them.

Alternative Implementation for PIPE

A. Two processors for the more expensive stage

There are three cases to consider in the execution of the pipe skeleton: more functions than free processors; equal number of functions and processors and more processors than functions. In the latter case the extra processors are liberated and therefore not used.

This alternative focuses on the situation mentioned above (more processors than functions) and proposes the use of an extra slave when the circumstances allow a significant improvement in the performance.

When all the stages of the pipeline have a similar time consumption, the performance of the system cannot be improved by adding another processor. But, what happen if one of the stages is significantly more compute intensive than
the others? (say twice or three times). The performance of the whole system would be given by the performance of this single stage. In other words, it would be the bottleneck of the system. The idea for avoiding this bottleneck is to use one of the extra slaves to perform the heavy stage in two parallel processors. Theoretically, sometimes it could be possible to decrease the response time of the pipe by 50% (if the communication time is not significant with respect to the processing time). The extra cost introduced by this alternative is the need for synchronisation between the two slaves which are executing the same function, in order to deliver the processed values in the correct order.

After solving this (first) bottleneck, it is possible to continue searching for other bottlenecks, and maybe adding more slaves duplicating other functions. It is also possible that one bottleneck could be solved by using more than two slaves performing the same stage of the pipe.

4.- IMPLEMENTATIONS FOR THE FARM SKELETON

FARM Original Implementation.

The farm skeleton receives as input two parameters: the function to be performed in each slave and a list of values to be processed. The output is a list of values processed by the farm.

As in the Pipe, the Farm execution can be divided in three sequential processes: Set-up, Process and Termination.

The Set-Up

This phase can be subdivided in two steps:

(1) Find out the slaves that are free at this point. An AREYOUFREE message is sent to every processor and waits for the replies, building a list with the free processors.

(2) An ALLOCATE message is sent to every free processor in the list and waits for the replies, building a list with the allocated processors. The function is included in the message.

The Process

The farm main execution can be described as an iterative process of sending and receiving messages from the slaves, until all the data is processed. In other words, the general idea is that the master sends the data to the slaves, one by one, and waits for the results.

The overall process of sending and receiving messages must be synchronised between the master and the slaves. If the master has to decide between sending data to a free slave and receiving a result from another slave, it will always give priority to the first, in order to increase the degree of parallelism.
Alternative Implementations for FARM

A.- Shared Environment

Here the idea is to keep the environment in a common shared location rather than send it to each slave. This can be implemented using a file that contains the environment, and maybe giving the slaves a function to access it. The key point is that the slaves read only that part of the environment they are interested in, and they do not need to load the whole of it in memory.

B.- Tree processing

The propagation of the farm function and the environment during the set up phase is done sequentially, from the master to each slave. Accordingly, we can propagate the data along the tree structure using the same principle. It is important to take into account that now the slaves will have more work to do. In addition to executing the farm function, each slave now has to communicate with other slaves: to look for one or more free slaves, to send them the data, to wait for their results and finally to send everything to its ancestor. In this solution some data will pass from the master to the 'processing' slave through other slaves. This means that one value could be included in more than just two messages, like in the original implementation, therefore the message passing can be increased considerably.

Alternative Implementations for both Pipe and Farm Skeletons

In this section are presented two alternative implementations which are suitable for both Pipe and Farm skeletons.

A.- Master Processing

The idea is to give more work to the master, in addition to its responsibility of distributing data and collecting results. Sometimes the master can be idle, when all the slaves are busy processing the data. So, why do not give the master some of the processing work?

B.- More data in each Message

The original implementations of pipe and farm send one value at a time to the slaves. The problem is that each message has associated a fixed cost, because of the header of the message, synchronisation time, etc. Hence, the idea is to allow the user to send more than one value in each message in order to reduce the communication costs whenever possible. The amount of data sent in each message is decided by the user.

5.- IMPLEMENTATIONS FOR THE DIVIDE AND CONQUER SKELETON

DIVIDE AND CONQUER Original Implementation

The Divide and Conquer skeleton can be divided into two phases: Set-Up and Process. It is important to emphasise that this skeleton is implemented on top of another skeleton, called parallel map (pmap). This skeleton works very similarly to the farm skeleton, but it does not involve an environment.
The Set-Up

This phase finds out the slaves that are free at this point. An AREYOUFREE message is sent to every processor and waits for the replies, building a list with the free processors.

The Process

The body of this skeleton works recursively until either the problem is decomposed to a trivial one or there are no more free processors, so the problem is solved locally using a sequential version of Divide and Conquer.

The process applies the parallel map skeleton over the decomposed problem (list of subproblems). On the other hand, the parallel map applies the Divide and Conquer process to each subproblem in the list, and so on.

Alternative Implementations for Divide and Conquer

A.- The master looks for trivial subproblems, and the slaves solve them.

One of the main issues to consider in the original implementation is that the skeleton parallel map is called as many times as decompositions take place. The problem with this is that the skeleton parallel map executes its set up each time it is invoked, and this includes sending messages to all the other slaves asking if they are free and then allocating the appropriate ones.

This alternative proposes to call the skeleton parallel map just once. For this purpose, it is necessary to find previously all the trivial problems on the master and then map the solve function over the list of trivial subproblems on the slaves. The decomposition of problems and the combination of results will be performed in the master and the solve function applied in the slaves.

In some sense the degree of parallelism is decreased because while the master is decomposing the problem and then combining the results the slaves are idle.

B.- Gradual distribution of the work

As mentioned above, one of the main problems of the original implementation is the continuous use of the parallel map skeleton. This new alternative presents another solution which simply gets rid of the use of the parallel map skeleton.

The following alternative, is focused on the point that after decomposing one problem, the master (or the slave which decomposed the problem) can continue applying the Divide and Conquer to one of the subproblems, after sending the others to the slaves.

6.- RESULTS

In this section we present the main results found in the performance comparisons made between the alternative implementations described in sections 3 to 5. The main objective is to identify the key factors that affect the performance of each implementation. This would support the development of performance models for the alternative implementations, and in a next stage,
the decision process of selecting the proper implementation for a particular application.

PIPE

For the Pipe skeleton were implemented two alternatives: (1) Two processors for the more expensive stage and (2) More than one value in one Message. The variables considered in the comparisons were the following: size of the problem (amount of data to be processed), cost (complexity) of the stages of the pipe, number of processors, number of stages in the pipe.

Case 1: Original Implementation versus Alternative (1). The alternatives of 2 and 5 values per message have been compared here. The pipe considered has 5 stages and all the functions are expensive. Alternative 1 (2 values per message) and the original implementation have similar performances, but the latter is slightly better than the first. The optimal point, as expected, is found for 6 processors (1 processor per stage and 1 processor for the master).

Comments for Pipe Implementations

* For pipelines with a higher number of stages (say 20), a sequential solution is not preferable. In this scenario, for simple functions the alternative implementation (1) is better.

* When the functions are expensive, the optimum point is obtained for (number of stages + 1) processors. But, sometimes it would be more convenient to have a lower performance and to use less processors.

* When there is a bottleneck in the pipeline and there are more than (1+ number of stages in the pipe) processors available alternative (2) is recommendable.

FARM

For the Farm skeleton was implemented one alternative: (1) Master doing some Processing. The variables considered in the comparisons were the following: size of the problem (amount of data to be processed), cost (complexity) of the functions, number of processors.

Case 2: Original Implementation versus Alternative (1). Two possibilities have been used for this alternative: master has 1 or 5 slaves in it. The function is expensive. Size of the problem 20. When the farm function is computing intensive, the communication time is not relevant, and therefore the optimal point is for 11 processors (1 for the master and 1 for each element to be processed). There is not much difference between the alternative implementation and the original one over 10 processors, because the master has no work to do.

Comments for Farm Implementations

* Alternative (1) focuses on the problem of message passing cost. Therefore, it is suitable only when the farm function is simple and the communication time is significant. But, if the number of elements to be processed is low it is recommended a sequential process.
Divide and Conquer

For the Divide and Conquer skeleton were implemented two alternatives: (1) Master looks for trivial subproblems and the slaves solve them and (2) Gradual distribution of the work. The variables considered in the comparisons were the following: size of the problem (amount of data to be processed), cost (complexity) of the functions solve, decompose and combine, number of processors.

Case 3: Original Implementation versus Alternatives (1) and (2). The function solve is expensive. Size of the problem 20. Clearly the original implementation is the worst here (for 8 or more processors) because it has less processors solving subproblems, which is now the key issue. Alternative (1) is better than alternative (2) because the combine and decompose functions are performed sequentially, which is better when they are cheap. It is important to emphasise that the optimal number of processors is one plus the size of the problem, considering that in alternative (1) one processor is allocated for the master which is just distributing the work. In the current case, 21 processors is enough to solve all the subproblems in one round of messages between the master and the slaves. Using more than 21 processors will produce more overhead, as the master communicates with every available processor.

Comments for Divide and Conquer Implementations

* For low cost functions, increasing the size of the problem at some point alternative (2) is preferable. When the solve function is expensive, alternative (1) is recommended. When the decompose and/or the combine function(s) is(are) expensive, the original implementation is recommended.

* When the problem is very time consuming (all the functions expensive), either the original implementation (when there is plenty of processors) or alternative (1) (when the number of processors is less or equal the size of the problem) is recommended.

* When the solve function is expensive, both alternatives reach their optimum point for (1+ size of the problem) number of processors.

7. CONCLUDING REMARKS

This paper is based on the concept of skeleton introduced by Darlington et al. [DAR92], and presents an extension Darlington's methodology for parallel functional programming. The main contribution of the extension presented is that it gives the programmer the possibility of choosing between alternative implementations for each skeleton.

We developed alternatives to the original implementations described by [DAR92]. The result of our work show that it is indeed possible to improve the performance of the skeletons by taking into account problem parameters such as: size of the data structures, type of communication between processors, bottleneck processes, granularity, number of available processors, complexity of the functions involved, etc.

In particular, we have shown that the original implementations for the Pipe and Farm skeletons suffer from communication overheads, which have been improved by our alternative implementations. For example, the second alternative implementation of the Pipe skeleton achieved improvements of
around 20% for just 50 values processed in the pipeline. On the other hand, the alternatives for the Divide and Conquer skeleton achieve improvements up to 75%, and it is possible to identify clearly the cases where each of them is preferable.

REFERENCES


Case 1: Pipe (time in secs).

Case 2: Farm (time in secs).

Case 3: D & C (time in secs).