A Survey of Cost Models for Algorithmic Skeletons

Mohammad M. Hamdan

September 13, 1999

Abstract

This report presents a survey of performance models for parallel algorithmic skeletons. First, higher-order functions (HOFs) are presented according to the modelled skeletons. Next, the corresponding parallel implementations (skeletons) for the HOFs are discussed with the performance models that were constructed for the skeletons.

1 Introduction

Effective portability depends crucially on the predictability of performance. Therefore, accurate performance models are required to predict the behaviour of a given skeleton before successfully porting the application that is currently using this skeleton. For the importance of performance models for algorithmic skeletons, this report aims to survey some of them.

2 Higher Order Functions for the Modeled Skeletons

This section presents briefly the HOFs used to describe the skeletons in Section 3. The HOFs are classified into two major classes: general and application specific HOFs.

2.1 General Higher Order Functions

The HOFs are presented in this section according to the classification used to present their corresponding skeletons which was taken from [4].

2.1.1 Recursively Partitioned

A recursively partitioned skeleton works by generating subordinate problems as a result of dividing the original problem. These generated problems will also be divided further in an attempt to solve them. Two well known algorithms that belong to this category are quick-sort and least-cost search. *Rabhi* [16] gave a HOF describing this skeleton:

```
rp$ ind solve divide combine prob
         | ind prob  = solve prob
         | otherwise = combine prob
                        (map (rp$ ind solve divide  combine)
                             (divide prob))
```

where:

- ind prob: returns true if the problem *prob* is indivisible.
- solve prob: solves an indivisible instance of a problem *prob*.
- divide prob: divides the problem *prob* into sub-problems.
- combine prob sols: combines the solutions *sols*.

Darlington et al [9] have presented a HOF for the divide and conquer (DC) skeleton. Similar to the recursively partitioned skeleton, the idea is to split larger tasks into sub-tasks and combine the results obtained by solving the sub-tasks independently. Their HOF for DC is as follows:

$$DC :: (\alpha \to Bool) \to (\alpha \to \beta) \to (\alpha \to [\alpha]) \to ([\beta] \to \beta) \to \alpha \to \beta$$

```
DC t s d c x | t x       = s x
             | not (t x) = (c . map (DC t s d  c) . d) x
```

where:

- t x: tests if a problem x is trivial.
- s x: solves a problem x.
- d x: divides a larger task x into sub-tasks.
- c ((map DC ...) x): combines the sub-results to produce the main result.

Cole [5] has also presented a HOF for his fixed degree divide and conquer (FDDC) skeleton. This is a special form of the general divide and conquer skeleton where the number of sub-problems to be divided are fixed in advance.

$$D_C : (prob \to bool) \to (prob \to [prob]) \to$$

$$([sol] \to sol) \to (prob \to sol) \to (prob \to sol)$$

```
D_C indivisible split join f = F
         where F P   = f P, if indivisible P
                     = join_k (map F (split_k P)), otherwise
```

2.1.2 Task Queue

Cole [5] has not presented a HOF for the *task queue* skeleton. However, a HOF for the *process farm* skeleton (discussed below) can be regarded as a special form of *task queue*. Algorithms that belong to *task queue* have workers in a work pool where each worker will check if tasks are available and then group the available one. Next, the workers will execute the tasks which might result in creating other tasks. The generated tasks will be added to the queue. The algorithm terminates when no more tasks are available and all processors are inactive. When tasks are executed they access a shared data structure that represents the problem to be solved. The *task queue* can have a number of queuing disciplines; stack, FIFO queue, unordered heap and strictly ordered queue.

Darlington et al. [9] have presented a HOF for the *process farm* skeleton which is a special form of the *task queue* skeleton. The *process farm* skeleton represents simple data-parallelism. A function is applied to a list of data. Parallelism arises by utilising multiple processors to apply the function to different parts of the list. Their HOF is as follows:

$$FARM :: (\alpha \to \beta \to \gamma) \to \alpha \to ([\beta] \to [\gamma])$$

```
FARM f env = map (f env)
```

2.1.3 Systolic

The *systolic* skeleton (which is the general form of pipeline-like skeletons) consists of a number of stages where parallelism is exploited by operating on a flow of data that passes through the various stages of the pipeline.

Darlington et al [10] have presented a HOF for the pipe skeletons, which is:

$$PIPE :: [[\alpha] \to [\alpha]] \to [\alpha] \to [\alpha]$$

```
PIPE = foldr1 ( o )
```

2.2 Application-based Higher Order Functions

There were no HOFs presented for the modelled application-based skeletons. Nonetheless, the performance models for the application-based skeletons are reported in Section 3.2.

3 Skeleton Implementations and Performance Models

This section briefly discusses the parallel implementations (skeletons) and performance models for the preceding HOFs.

It is important to note that for a given HOF, it is possible to have more than one parallel implementation. Also, a parallel implementation can be used to implement a number of HOFs. Similar to the presentation of the HOFs, the skeletons are divided into two major classes: general and application-specific skeletons.

3.1 General Skeletons

Campbell's [4] general classification of algorithmic skeletons is used to present the performance models for the general skeletons. The categories in his classification are discussed in the following sections. In each section, the first skeleton discussed was found to be the most general skeleton of all similar skeletons according to his classification.

3.1.1 Recursively Partitioned

Rabhi's recursively partitioned skeleton was implemented using *distributed graph reduction* [16]. However, no performance models were developed for this skeleton as far as we know as the author's concern was to provide a link between meaning and behaviour. Nonetheless, other performance models for skeletons that belong to this category are discussed below.

The DC skeleton for *Darlington et al's* [9] DC HOF was implemented using the functional language $Hope^+$ as the source language and using C as the target language. The CS Tools library was linked to the C code to provide the communication layer on two platforms: a Meiko Transputer Surface and a Fujitsu AP1000. They also developed a performance model for their DC skeleton. Their general model still needs to be instantiated with average communication and computation costs measured from the target implementation. The DC performance model was not developed further by instantiating it with measurements taken from the target implementation. However, performance models for other skeletons were instantiated with the measurements and this will be mentioned in Sections 3.1.1 and 3.1.2.

The performance model for DC assumes the processors are organised into a balanced binary tree and all processors will eventually be used as leaves (i.e intermediate nodes also perform work). The execution time for DC can be predicted using this formula:

$$t_{sol_x} = \sum_{i=0}^{\log(p)-1} (t_{div_{x/2^i}} + t_{setup_{x/2^i}} + t_{comb_{x/2^i}} + t_{comm_{x/2^i}}) + t_{seq_{x/2^{\log p}}}$$

where t_{sol_x} is the time to solve a problem of size x, t_{div_x} is the time to divide a problem of size x, t_{comb_x} is the time to combine the two results, t_{setup_x} and t_{comm_x} are setup and transmission time for communication and t_{seq_x} is the time to solve a problem of size x sequentially.

Cole [5] did not have a parallel implementation for the FDDC HOF shown in Section 2.1. He only showed how an idealised implementation would be. In this case, a k-ary tree of processes is mapped onto a k- ary tree of processors. The root processor will divide the original problem into k sub-divisions. This sub-division process will carry on recursively until no more sub-divisions are possible. When this happens, the leaf processors will compute the base function *f*. Finally, the results will be passed up the tree to the root processor which will compute the result for the whole problem.

Asymptotic performance analysis was the technique used to generate a formulae to expect the execution time for this skeleton. The formulae that was given to describe the time complexity of the H-tree [1] implementation of the binary FDDC skeleton is as follows:

$$T_{P2}(n) = 4(\sqrt{n} - 1) + f(1) - 2\log n + \sum_{i=1}^{\log n} [s(2^i) + j(2^i) + p(2^{i-1}) + r(2^{i-1})]$$

when the depth $\log n$ is odd the first term in the above formulae is replaced by $3\sqrt{(2n)} - 4$. Where T_{P2} is the parallel run-time for a binary tree implementation, $s(n)$ is the time to split a problem of grain n into two sub-problems of grain $\frac{n}{2}$, $p(n)$, $r(n)$ are the number of constant size packets of information required to describe a problem instance and result of a problem of grain n respectively, $j(n)$ is the time to join two sub-results of grain $\frac{n}{2}$ to form one result of grain n and $f(1)$ is the time needed to evaluate the base case function on an indivisible instance. The above model is simply the summation of three phases: 1) splitting and distributing the problem, 2) computing the base case at the leaves and 3) returning and combining the results. However, the above performance model was not verified on any target architecture but it inspired other researchers in the field to work on performance models for skeletons.

[1] The H-tree implementation is a well known layout of a complete binary tree on a grid of processors.

Although, *Feldcamp et al* [8] did not present a HOF for the DC skeleton they developed. Their skeleton was integrated in the program development environment Parsec [12]. The DC was implemented as a binary tree on a Transputer network. This version of DC is similar to *Cole's* FDDC skeleton except that internal processors of the tree topology also perform work.

The performance model for the DC skeleton which was developed by Sreekantaswamy [20] for speedup with startup cost taken into account is as follows:

$$SP(N) = \frac{MT_e}{T_{startup} + \frac{M-1}{S_N}}$$

where steady state throughput is

$$S_i = S_{i-1}\left(\frac{T_e(i) + \beta_e - T_s(i) - T_j(i) - \beta_f}{T_e(i) + \beta_e}\right) + \frac{1}{T_e(i) + \beta_e}$$

and startup time is

$$T_{startup} = \sum_{i=2}^{N}(2T_\tau(i) + T_s(i) + T_j(i) + \beta_f) + T_e(1) + \beta_e$$

For the preceding models, M is the number of tasks, $T_s(i)$ is the CPU time for splitting a task at level i, $T_j(i)$ is CPU time for joining tasks at level i, $T_e(i)$ is CPU time for executing a task at level i, $T_\tau(i)$ is the transfer time of a task at level i, β_e and β_f are the CPU overhead for processing a task locally and forwarding a task, respectively. For example, β_f on a Transputer could be the CPU overhead resulting from the link setup time which cannot be overlapped with computation. The startup time is simply the time needed to return the first result which is affected by the scheduling strategy. The startup time can be neglected in cases where there are large number of tasks and in turn large execution time, otherwise it is necessary to consider it. The steady state throughput S_N of the complete tree can be calculated by recursively evaluating the S_i's up to level N. This performance model differs from other models as it includes the CPU overhead resulting from non-overlapped start-up cost of communication. The performance model was validated on a 74 node T800 Transputer based multicomputer system. The difference between the measured speedup and predicted speedup by the model was claimed it did not exceed 5%.

Campbell [4] has developed a model for parallel computation called CLUMPS. CLUMPS stands for *Campbell's Lenient, Unified Model for Parallel Systems* which is composed of an abstract machine with an associated cost model. For completeness, the modelling of DC under CLUMPS is presented below which is an asymptotic analysis of DC:

$$O(\left(\frac{N}{P}SOLVE\right) + \left(\log P + 2\frac{N}{P} - 1\right)TEST + \left(\log P + \frac{N}{P} - 1\right)(ORDIVIDE$$

$$+ANDCOMBINE) + f(\log P, d(N)) + f(logP, c(N)))$$

where N is the problem size, P is the number of processors, $SOLVE$ is the cost of solving an indivisible problem of size 1, $TEST$ is the cost of testing whether a problem is indivisible, $ORDIVIDE$ is the cost of dividing a problem into 2 sub-problems, $ANDCOMBINE$ is the cost of combining 2 sub-solutions to sub-solutions, $d(N)$ is the number of communication packets required to represent a problem of size N, $c(N)$ is the number of communication packets required to represent a solution of size N, $f(\log P, d(N))$ is the communication cost and $f(\log P, c(N))$ is the communication cost for re-combination.

3.1.2 Task Queue

Similar to the FDDC skeleton, *Cole* [5] has showed how an idealised implementation would be for the *Task Queue* skeleton on a 2D grid. Asymptotic analysis was used to describe the time complexity to route tasks across the grid.

Darlington et al's [9] *FARM* skeleton was implemented in a similar way to their DC skeleton discussed in Section 3.1.1. The *FARM* consists of two major parts: 1) a master processor and 2) a worker processor(s). The master processor has a pool of work packets that need to be processed. Therefore, it distributes them to workers. Once the worker receives a work packet it will process it to compute the result. Next, it sends the result back to the master processor. The work finishes when the master processor receives back all the results. In summary, the *FARM* work using a "distribute-work-collect" phase. The number of steps needed to complete the work depends on the number of work packet and number of workers.

In the beginning, a general performance model was developed for the *FARM* skeleton:

$$t_{farm} = t_s + R(t_e + 2t_c)$$

where t_s is the skeleton startup overhead, R is the number of steps required to process the work pool which is simply $\lceil \frac{N}{P} \rceil$, P is the number of workers and N is the number of work packets, t_e is the required time to solve a work packet and t_c is the communication time required to receive a result or send the data.

The above model was developed further by instantiating it with measurements taken from a Transputer implementation. The machine parameters used to instantiate the general performance model for the *FARM* skeleton were: 1) average execution time of instructions on each of the processors, 2) number of processors used by the skeleton and 3) communication costs of the machine.

The communication time t_c can be refined in terms of the communication startup time to send or receive one byte of data t_b. In this case $t_c = t_o + t_b b$ where b is the number of bytes to transfer. The final performance model for the process farm was as follows:

$$\begin{aligned}
T_{farm} &= T_{setup} + R(T_{comms} + T_{slave}) + T_{gc} \\
\\
where & \\
T_{setup} &= I_o + I_1 P \\
R &= \left\lceil \frac{N_w}{g(P-1)} \right\rceil \\
T_{comms} &= 2T_o + g(d+r)T_b + g(2T_{fo} + T_f(r+d)) \\
T_{slave} &= gT_w \\
T_{gc} &= \left\lfloor W_s g G \left\lceil \frac{N_w}{g(P-1)} \right\rceil \right\rfloor T_g
\end{aligned}$$

where T_{setup}, is the setup time taken to allocate processors to a skeleton can be determined by benchmarking once for each skeleton. The constants I_o and I_1 depend on the initialisation protocol used. The number of rounds, R, depends on the number of work packets N_w, the clustering unit g and number of processors P. The communication time, T_{comms}, can be refined in terms of the communication startup overhead and the time taken to send one byte on the network between two processors denoted by T_o and T_b respectively. The process of *flattening*[2] and *unflattening*[3] the data structures used by $Hope^+$ imposed further overheads on communication. The two parameters for this overhead are T_{fo}, flatten startup overhead, and T_f, time to flatten, which are measured by benchmarking. The variables d and r represent the size of the distributed work packets and the corresponding results, respectively. The time to process a work packet on any of the slaves, T_{slave} is the result of multiplying the clustering unit, g, by the time required to process a work packet, T_w. An additional overhead is added to the above formulae which results from garbage collection, T_{gc}. This overhead depends on the amount of work done for each packet, W_s, the frequency at which garbage collection occurs, G, and the average time taken for a call to the collector T_g. Experimental results showed the accuracy of the performance model in predicting the *FARM* behaviour for the Transputer implementation.

Sreekantaswamy [20] has also developed a performance model for the processor farm skeleton. In the model it was assumed that tasks do not split, processors choose to execute the task or pass it to a child processor and for all processors i, $T_e(i) = T_e$, where T_e is the processing time of the task. The performance model for the speedup with startup costs taken into account is as follows:

$$SP(N) = \frac{MT_e(1-a^N)}{(M-1)(T_e+\beta_e) - k(T_e+\beta_e-\beta_f) + T_{startup}(1-a^N)}$$

[2] converting data structures to byte streams.

[3] converting byte streams to data structures.

where

$$a = k(T_e + \beta_e - \beta_f)/(T_e + \beta_e),$$

steady-state throughput:

$$S(N) = \frac{1}{(T_e + \beta_e) - k(T_e + \beta_e - \beta_f)} \left[1 - \left(\frac{k(T_e + \beta_e - \beta_f}{T_e + \beta_e}\right)^N\right]$$

and start-up time:

$$T_{startup} = (N-1)(2T_\tau + \beta_f) + T_e + \beta_e$$

The variables in the above formulae have the same meaning as the performance model for the divide and conquer skeleton discussed in Section 3.1.1. The model assumes the processor farm structured as an N level balanced k-ary tree executing M tasks with average processing time T_e. The startup time for this skeleton is simply the time it takes to forward the first task to the Nth level of the tree and return the computed result. Note that the startup overhead for forwarding a task, β_f, was subtracted from the $(T_e + \beta_e)$ in the steady state throughput formulae as it was already added in the startup formulae for (N-1) processors. The performance model was validated on a 74 node T800 Transputer based multicomputer system. The difference between the measured speedup and the predicted speedup by the model was claimed it did not exceed 5%.

Bratvold [2] has also developed a performance model for the FARM skeleton. The completion time for the skeleton was given as follows:

$$\begin{aligned}
T_C &= T_{start} + T_{run} + T_{close} \\
& \\
where & \\
T_{start} + T_{close} &= P_w * Farm_{startclose_add} + Farm_{startclose_base} \\
P_w &= P - 1 \\
T_{run} &= max(T_{run1}, T_{runP_w}) \\
T_{run1} &= (n-d) * (t_f + t_k) + (N - (n-d)) * t_p \\
T_{runP_w} &= (n+d) * (t_f + t_k) \\
d &= \frac{t_p * (N - n)}{2 * t_f + 2 * t_k - t_p} \\
t_p &= t_{ep} + t_{cp} \\
t_k &= t_{ek} + t_{ck}
\end{aligned}$$

where T_{start} and T_{close} are for the time when not all workers are busy, i.e idle time which is the extra overhead, T_{run} is the time when all workers are busy working, P_w is the number of workers $(P-1)$, $FARM_{startclose\_base}$ is a fixed cost when some workers are idle, $FARM_{startclose\_add}$ is the additional overhead cost for all workers in the $FARM$, T_{run1} and T_{runP_w} are the total computation times for the first and last workers respectively, n is the average number of elements that each worker gets which is equal to $\frac{N}{P_w}$, d is a constant, t_f is the execution time for function f, t_p is the time for passing the data elements to other workers and results towards the farmer, t_k is the time for receiving elements and keeping them in local memory and sending the results from local memory to the farmer, t_{ep}, t_{cp}, t_{ek} and t_{ck} are the times resulting from communication overhead. The validation of the performance model for the Transputer implementation had an accuracy of around 90%.

The model was designed specifically for the Transputer implementation and may not work on other parallel architectures. Also, the performance model was for a linear list process farm. However, it may be possible to modify the model and make it more general then instantiate it with machine parameters for other architecture similar to *Darlington et al's* performance models.

Busvine's [3] performance model for the *FARM* skeleton is as follows:

$$T_{Farm} = max(N * (T_{Rec} + K_{Rec} + T_{Comms}) + T_{Base},$$
$$(N/W) * (T_{Task} + K_{Worker} + K_{Tree} * T_{Comms}))$$
$$+N * (T_{Combine} + K_{Combine}) + K^{W}_{Setup}$$

where T_{Rec} is the average time for the farmer to generate a task for each level in the recursion, K_{Rec} is the time to store tasks for later forwarding to workers and saving the results of evaluated tasks received from the workers, T_{Comms} is the time for communicating one task and its result, T_{Base} is the time for evaluating a single base case, T_{Task} is the average time for task evaluation, K_{Worker} is a constant time for unpacking tasks and packing results ready for communication, K_{Tree} is a constant that depends on the shape of the worker pool, $T_{Combine}$ is the time for combining the evaluated tasks, $K_{Combine}$ is the time for extracting an evaluated task, K^{W}_{Setup} is the time for starting up and shutting down the worker pool, N is the depth of recursion and W is the number of workers.

The process farm was implemented as a tree with the farmer as root. Values for the constants K_{Rec}, K_{Worker}, K_{Tree} and $K_{Combine}$ were tuned by fitting this equation for T_{Farm} against measured running times.

From the parameters used in the model like T_{Rec} and N, we can note that the model was used for a special purpose. The model was used mainly to determine whether a linear recursive function should be implemented as a process farm. Therefore, the model has a limitation in predicting the performance for general uses of the process farm skeleton.

3.1.3 Systolic

Darlington et al [10] have developed a performance model for the pipeline skeleton which is:

$$t_{pipe} = t_s + (t_e v + t_c)(p + n - 1)$$

The above performance model resembles a general model for the pipe skeleton which predicts the total execution time of a particular instance of the pipe skeleton. Where t_s is the startup time, t_c is the execution time of a stage for a single element, t_c is the communication time between stages, p is the number of stages and n is the number of elements in the list.

Similar to the process farm skeleton, the pipe performance model was further instantiated with measurements taken from the Transputer implementation.

$$\begin{aligned}
T_{pipe} &= T_{setup} + T_{fill} + (R-1)(T_{prc} + T_{comms}) + T_{gc} \\
\\
\mathit{where} & \\
R &= \left\lceil \frac{N_w}{g} \right\rceil \\
T_{prc} &= gT_w \left\lceil \frac{N_p}{P-1} \right\rceil \\
T_{fill} &= P(T_o + dT_b) + (P-1)(2(T_{fo} + dT_f) + T_{prc}) \\
T_{comms} &= T_o + dT_b + 2(T_{fo} + dT_f)
\end{aligned}$$

For this model, N_w is the total number of work packets, T_w is the time to process one packet, g is the clustering unit, d is the number of bytes transferred between stages for each packet and N_p is the total number of virtual stages [4] which is independent of the number of processors. T_{prc} is the processing time, T_{comms} is the communication time, T_{fill} is the time to fill the pipe, T_{gc} is the time for garbage collection and T_{setup} is the setup time. The experimental results showed less accurate predictions for the pipe skeleton comparing it with the FARM skeleton. This was due to the poor modelling of the garbage collector.

Bratvold [2] has also developed a performance model for the pipeline skeleton. However, he assumed a pipeline of s stages where each stage is a skeleton. The total latency[5] of the skeleton was given by

$$L = \sum_{i=1}^{s} (L_i + c_i)$$

[4] In an ideal pipe it is assumed that one processor was allocated to each process. However, this is not always possible due to limited resources. Therefore, a number of virtual stages, need to be allocated to a real stage.

[5] The latency of a skeleton is the time taken from the first of the input list is fed into a skeleton until the first result element appears.

where L_i is the latency of stage i, and c_i is the time to communicate one data element from stage i to stage $i+1$ assuming $c_s = 0$ as s is the last stage. The completion time is given by

$$T_C = (max_{i=1}^{s} T_{C_i}) + L - L_{max}$$

Where each stage i has a completion time of T_{C_i} and L_{max} is latency of the stage with highest T_{C_i}.

3.2 Application-based Skeletons

Skeletons in this category were primarily developed for use in specific parallel programs. Normally, program developers who use such skeletons are experts of the problem field of research. In other words, knowledge is required for using such skeletons.

Deldarie et al [13] have presented performance models for special cases of skeletons related to image processing. One skeleton called *local window* (LW) which transforms an image into another image. Each pixel in the resulting output is derived from pixels in a window surrounding the corresponding pixel in the input image. Another skeleton called *split and merge* (SAM) where an image is partitioned into slices, an operation is applied to each slice then the results are merged. This skeleton resembles the divide and conquer skeleton.

The overall time complexity for the LW skeleton is:

$$O(\log^2 P + t_w n^2/P)$$

Where $O(\log^2 P)$ is the cost for forking and joining P processes, t_w is the cost for a single window function, n^2 is the number of pixels in image and P is the number of processors. For the SAM skeleton the overall time complexity is:

$$O(\log^3 P + n^2/P + t_s(n^2/P) + \sum_{i=1}^{\log P} (I_i + t_m(I_i) + J_i))$$

Where $\log^3 P$ is the result of multiplying $\log P$ merge states with the cost for barrier synchronisation costing $\log^2 P$, t_s is the cost for solving a slice of n^2/P pixels, t_m is the cost of merge,I_i is the number of bytes copied from global memory and J_i is the number of resulting bytes from the merge to be copied to global memory.

The execution time for the SAM and LW skeletons was presented as a *generic higher order complexity function.* Also, the models were extended to cover scalability analysis using the *isoefficiency function* [6].

4 Cost Models for Skeleton-Oriented Programming

The following researchers: S. Pelagatti [15], R. Rangaswami[17], D. Skillicorn [18] and H. W. To [21] have developed cost modelling systems in conjunction with their models for skeleton-oriented programming. Before outlining the cost models they have developed for their skeletons, we need first to present their methodologies for skeleton-oriented programming. The skeletons in this section differ from the previous skeletons in that they are integrated into one system for skeleton-oriented parallel programming.

S. Pelagatti [15] has participated in developing an explicit parallel programming language called P^3L [11, 7]. She has developed cost models for the pipe and process farm skeletons.

R. Rangaswami [17] has developed a parallel programming model called **HOPP** for skeleton-oriented programming. **HOPP** stands for Higher-Order Parallel Programming. The model consists of three parts: the *program* model, the *machine* model and the *cost* model. The program model is a composition of nested instantiations of recognised and user-defined functions [6]. Each stage in the pipeline resulting from the composition is referred to a *phase* of the program. The machine model targets architectures for the programs which include hypercube, 2-D torus, linear array and tree. The cost model determines the cost of executing a recognised function on a given architecture.

A **HOPP** program could comprise of one or more *phase(s)* where parallelism is exploited only within each *phase*. The parallelism exploited in each phase depends on the number of recognised functions in the phase itself. For example, if there are n recognised functions in a given phase then it is possible to have $2^n - 1$ different parallel implementations for that phase. In other words, there is parallelism if there are recognised functions. However, parallelism was exploited only for the first three recognised functions. Any recognised functions below the first 3 was implemented in parallel.

The cost models for the recognised functions were used to statically analyse a given **HOPP** program to determine the best cost-effective parallel implementation for a given architecture. Actually, the theoretical cost models were implemented as an analyser. In general, the cost of a program comprising of n *phases* is given by:

$$Cost = \sum_{n}^{i=1} C_{pi} + \sum_{i=0}^{n-1} C_{i,i+1}$$

where C_{pi} is the cost of phase i and $C_{i,i+1}$ is the communication cost in case there is a need for rearranging the output of phase i to suit the implementation of phase $i+1$.

[6] All of the recognised functions work on lists.

In the implementation scheme, five modes of communication were identified for rearranging the data between the phases:

- Nearest Neighbour Communication: in this operation a processor sends a message to another processor which has a direct link with it.
- Broadcast Operation: in this operation a processor sends a message to all the other processors in the network.
- Scatter Operation: in this operation a processor will move different messages to all the processors in the network.
- Gather Operation: in this operation a processor will collect the distributed data packets across the processors in the network.
- Total Exchange Operation: in this operation data packets from every processor will be moved to every other processor in the network.

Next we outline the cost models of the recognised functions for the supported topologies. The models are grouped according to the topology they support. First we describe the meaning of each variable used in the cost models.

- n: is the list size.
- p: is the number of processors.
- d: is the dimension of the hypercube.
- C_f: is the cost of the processing function.
- T_{com}^m: is the cost of communicating m elements of the list to a neighbour. It is evaluated by $T_{com}^m = K_0 + \frac{1}{K_1} ms$, where s is the size in bytes of each element of the list, K_0 and K_1 are the startup cost to initiate the communication and the cost to transfer data, respectively.
- C_r: is the cost of reversing the portion of the list on each processor.
- G: is the cost of gathering n elements distributed across p processors on the particular topology.
- C_{be}: represents the cost of Broadcast or Total Exchange.
- B: is the cost for broadcasting m elements to p processors, on the particular topology.
- C_i: is the cost of phase i including and data rearrangement costs.
- $C_{PAR\_FOLD}^{plus}$: is the cost of the **fold** operation.

skeleton	cost model
s_fold	$C^{a}_{s\_fold} = C_f(\frac{n}{p} - 1 + \log p) + T^{1}_{com}(p-1)$
g_fold	$C^{a_c}_{g\_fold} = C_f(\frac{n}{p} - 1 + \log p) + \sum_{i=0}^{\log p - 1} 2^i T_{com}^{2^i \frac{n}{p} m}$
s_scan	$C^{a}_{s\_scan} = C_f(2\frac{n}{p} + p - 4) + T^{1}_{com}(p-1)$
g_scan	$C^{a_c}_{g\_scan} = C_f(2\frac{n}{p} + p - 4) + \sum_{i=1}^{p-1} T_{com}^{i \frac{n}{p} m}$

Table 1: Cost models of skeletons for the linear array topology.

skeleton	cost model
s_fold	$C^{h}_{s\_fold} = C_f(\frac{n}{p} + d - 1) + T^{1}_{com} d$
g_fold	$C^{h\_c}_{g\_fold} = C_f(\frac{n}{p} + d - 1) + \sum_{i=0}^{d-1} T_{com}^{2_i \frac{n}{p} m}$
s_scan	$C^{h}_{s\_scan} = C_f(2\frac{n}{p} + 2d - 4) + T^{1}_{com}(2d-1)$
g_scan	$C^{hc}_{g\_scan} = C_f(2\frac{n}{p} + 2d - 4) + \sum_{i=0}^{d-1} T_{com}^{2^i \frac{n}{p} m} + (d-1) T_{com}^{2^{d-1} \frac{n}{p} m}$
reverse	$C^{h}_{reverse} = C_r + 2T_{com}^{\frac{n}{p}}$

Table 2: Cost models of skeletons for hypercube topologies.

- C_{lproc}: is the list processing costs.
- B_1: is the cost of broadcasting a single element of the list to all the processors.

For the skeletons **fold** and **scan** there two versions: a static and growing. For the static version, the emerging result remains constant in every step. However, for the growing version the size of the emerging result grows by an amount equal to the size of each list element.

Linear Array:

Table 1 summarises the cost models of the skeletons for the linear array topology.

Hypercube:

Table 4 outlines the cost models of the skeletons for the hypercube topology.

2-D Torus:

Table 4 outlines the cost models of the skeletons for the 2-D torus topology. Also, cost models were developed for the tree topology.

All Topologies

Table 4 outlines the skeletons which have one cost model for the four topologies.

D. Skillicorn et al [18, 19] have developed a *cost calculus* for the Bird-Meertens Formalism (BMF) [1]. First we briefly describe BMF before describing

skeleton	cost model
s_fold	$C^{m\_ideal}_{s\_fold} = C_f(\frac{n}{p} + 1 + \log\frac{p_1}{2} + \log\frac{p_2}{2} + T^1_{com}(\frac{p_1+p_2}{2})$
g_fold	$C_f(\frac{n}{p} + 1 + \log\frac{p_1}{2} + \log p_2 2) + \sum_{i=0}^{\log\frac{p_1}{2}-1}(2^i T_{com}^{2^i\frac{n}{p}m}) +$ $T_{com}^{\frac{p_1}{2}\frac{n}{p}m} + \sum_{i=0}^{\log\frac{p_2}{-}1}(2^i T_{com}^{2^i\frac{n}{p_2}m}) + T_{com}^{\frac{n}{2}m}$
s_scan	$C^m_{s\_scan} = C_f(2\frac{n}{p} + p_1 + p_2 - 4) + T^1_{com}(\frac{3}{2}p_1 + p_2 - 1)$
g_scan	$C^{mc}_{g\_scan} = C_f(2\frac{n}{p} + p_1 + p_2 - 3) + \sum_{i=1}^{p_1-1} T_{com}^{i\frac{n}{p}m} +$ $\sum_{i=1}^{p_2-1} T_{com}^{i\frac{n}{p}mp_1} + \frac{p_1}{2} T_{com}^{\frac{n}{p}mp_1(p_2-1)}$

Table 3: Cost models of skeletons for 2-D torus topologies.

skeleton	cost model
map	$C_{map} = \frac{n}{p} C_f$
filter	$C^p_{filter} = C_f\frac{n}{p} + G$
inits	$C^p_{inits} = C_{be}$
cross_product	$C_p^{cross\_product} = B + C_f\frac{n}{p}m$
composition for k phases	$C_{composition} = \sum_{i=1}^{k} C_i$
split	$C^p_{split} = S_{\frac{n}{p},\frac{k}{p}}$
len	$C^{len}_p = C_{subst\_1}(\frac{n}{p} - 1) + C^{plus}_{PAR\_FOLD}$
copy	$C^p_{copy} = C_{lproc} + B_1$

Table 4: Cost models for a number of recognised functions.

its cost calculus. BMF was developed by Richard Bird and Lambert Meertens which is a notation for describing functions with a calculus for manipulating function descriptions. The idea of BMF is to derive programs from problem specifications. However, most of the work on BMF has concentrated on lists. A theory in BMF describes the behaviour of a data type, and provides a set of operators on that type in addition to transformational laws for the operators. For example, operators for the list type include: map (written $\star$), filter (written $\triangleleft$) and reduce (written $/$). They can be used in the following way

$$\begin{array}{rl} f \star [a, b, c, ..., z] = [f\,a, f\,b, f\,c, ..., f\,z] & \text{(map with f)} \\ \oplus/[a, b, c, ..., z] = a \oplus b \oplus c \oplus ... \oplus z & \text{(reduce with } \oplus\text{)} \\ p \triangleleft [a, b, c, ..., z] & \text{(filter with } p\text{)} \end{array}$$

where f is a unary function and $\oplus$ is a binary function. Some of the transformation laws which can be applied to generate efficient programs include:

$$\begin{array}{rl} (f \circ g)\star = (f\star) \circ (g\star) & \text{(map distributivity)} \\ (p\triangleleft) \circ (q\triangleleft) = (q\triangleleft) \circ (p\triangleleft) & \text{(commutativity of filters)} \end{array}$$

Skillicorn has supported the use of BMF for parallel programming as it abstracts from parallel architectures which allows portability of programs and its operators like map and reduce contain implicit parallelism. They have defined a cost calculus to be a cost system with two properties: (1) the cost of a program depends on the cost of its pieces, i.e. compositional and (2) the cost of transformation can be associated with its rule.

The cost calculus that they have developed for BMF integrates cost information with equational rules. This feature adds the possibility in deciding the direction in which an equational substitution is cost-reducing. To simplify, the cost calculus will be used to guide the application of transformational rules, in this case it is for BMF transformational rules where equational rules will be used for transforming one parallel structure into another in an attempt to find a cost-effective implementation.

The cost of implementing the **map** function in parallel is:

$$\begin{array}{rcl} t_n(f\star_n) & = & t_n(PAR_n f) \\ & = & t_1(f) \end{array}$$

where $t_n(f)$ is the cost of executing f on n processors and $PAR_n f$ denoted n parallel applications of f.

The cost of a parallel reduction with constant-sized partial results is:

$$\begin{aligned} t_n(\oplus/_n) &= t_n(PARIMPL_n(\oplus/)) \\ &= \log n \end{aligned}$$

where $PARIMPL_n(\oplus/)$ is the parallel implementation of $\oplus/$ using n processors.

After finding the costs of the basic operations, the cost information was used to label BMF identities. For example, we have $(f \circ g)\star = f \star \circ g\star$ as a BMF identity. To label it we need to find the costs of the left hand side and the right hand side. For this identity both sides do have equal costs (cost-neutral). Therefore, we can label it in the following way:

$$(f \circ g)\star \rightleftharpoons f \star \circ g\star$$

Other identities were labeled in the same way and used to guide transformations for cost-effective implementations.

H. W. To [21] has looked into optimising the combinations of algorithmic skeletons. A language for combining skeletons was proposed and a set of *primitive skeletons* was chosen. The primitive skeletons were based on the operators of parallel abstract data types (PADTs)[7]. This choice was based on the observation that many highly parallel applications exploit parallelism from a large data structure. Two PADTs were described: restricted lists (*rlist*s) and *arrays*. A number of combining skeletons were proposed where each skeleton captures a common pattern of control flow. For example, the compose ($\circ$) function [8] which takes the output of one skeleton and pass it as the input of anther skeleton. Another two combining skeletons are **iterateFor** and **iterateUntil** which actually extend composition where the same skeleton might iterate over a data structure. Also it is possible to combine binary skeletons using the compose-2 ($\circ\circ$) function. In cases where it is necessary to apply more than one skeleton to an instance of a data structure the split ($\triangle$) skeleton could be used.

For each primitive skeleton two kinds of data-level parallelism were exploited: data distributed and data pipelined. In other words, how would the skeleton behave in case it is implemented using a data distributed or data pipelined parallelism?

For data distributed array primitives two distribution patterns were studied: *blocked* and *cyclic*. An example will explain techniques. Suppose the distribution of an array on p PEs. In a *blocked* distribution each processor will get a contigoues part of the original array. In a *cyclic* distribution, elements of the array will be distributed in a round robin style across the PEs. The idea of having

[7]A PADT is an aggregate data structure and a set of parallel operators with which to manipulate it.

[8]Note that we are talking about composing skeletons here and not functions.

two distribution is to a give trade-of between load balancing and communication overhead. Below we describe most of the performance models that were developed for *rlist* and *array* primitives. The meaning of variables used in the models will be explained only for the the first model as they have the same meaning for the other models.

Data distributed array primitives:

The performance model for the **aMap** skeleton is:

$$T_{aMap} = (n/p)t_e$$

where t_e is the time to compute an element, n is the size of the array and p is the number of processing elements. It can be noticed easily from the model that is does not include time for communication overhead. This is due to the static distributed implementation of the *aMap* skeleton.

The overall time for a blocked **aFolda** [9] is

$$\begin{aligned} T_{afoldaB} &= t_{leaf} + tyb + txb \\ &\text{where} \\ t_{leaf} &= \left(\frac{n_x n_y}{p_x p_y} - 1\right) t_e \\ t_{yb} &= \lceil \log p_y \rceil (t_e + 2T_{hand} + T_c M) + T_c(2^{\lceil \log p_y \rceil} - 1) \\ t_{xb} &= \lceil \log p_x \rceil (t_e + 2T_{hand} + T_c M) + T_c(2^{\lceil \log p_x \rceil} - 1) \end{aligned}$$

where t_{leaf} is the time for sequential reduction, t_{yb} is the reduction time along the y-axis, t_{xb} is the reduction time along the x-axis, the size of the array is (n_x, n_y) with $p_x p_y$ PEs, $\lceil \log p_y, \log p_x \rceil$ is the height of the reduction tree, t_e is the time for computing one element, T_{hand} is the cost of message startup, T_c is the channel cycle time which is the time required to transfer a unit of data across a hop and M is the size of an element which is assumed to be constant. Note that the communications cost is specific to architectures which use worm-hole routing.

The overall time for a cyclic **aFolda** is:

$$\begin{aligned} T_{afoldaC} &= t_{root} + t_{yc} + t_{xc} \\ &\text{where} \\ t_{yc} &= \lceil \log p_y \rceil \left(t_e \frac{n_x n_y}{p_x p_y} + 2T_{hand} + T_c M \frac{n_x n_y}{p_x p_y}\right) + T_c(2^{\lceil \log p_y \rceil} - 1) \\ t_{xc} &= \lceil \log p_x \rceil \left(t_e \frac{n_x n_y}{p_x p_y} + 2T_{hand} + T_c M \frac{n_x n_y}{p_x p_y}\right) + T_c(2^{\lceil \log p_x \rceil} - 1) \end{aligned}$$

where variables have the same meaning as in the performance model for **aFolda**. The difference is only t_{root} which is needed as the sequential reduction will be

[9] For an associative aFold.

done at the root. However, the completion time for cyclic distribution takes $(\lceil \log p_x \rceil + \lceil \log p_y \rceil - 1)$ times longer than block distribution.

Data distributed rlist primitives:

The cost for a data distributed **rMap** is:

$$T_{rMap} = t_e(n/(p_x p_y))$$

where t_e is the constant execution time, with a list of length n and $p_x p_y$ PE array.

The overall execution time for the *blocked* case of **rFolda** is:

$$\begin{aligned} T_{rfoldaB} &= t_{leaf} + t_{yb} + t_{xb} \\ \text{where} & \\ t_{leaf} &= \left(\frac{n}{p_x p_y} - 1\right) t_e \end{aligned}$$

the cost for performing the column reductions is:

$$t_{yb} = \begin{cases} \lceil \log p_y \rceil (t_e + 2T_{hand} + T_c M) + T_c(2^{\lceil \log p_y \rceil} - 1) & \text{if } p_x \text{ is even} \\ \lceil \log(p_y - 1) \rceil (t_e + 2T_{hand} + T_c M) + T_c(2^{\lceil \log(p_y-1) \rceil} - 1) + & \\ t_{extra_B} & \text{if } p_x \text{ is odd} \end{cases}$$

$$t_{extra_B} = t_e + 2T_{hand} + T_c(M+1))$$

and the cost of reducing the rows is:

$$t_{xb} = \lceil \log p_x \rceil (t_e + 2T_{hand} + T_c M) + T_c(2^{\lceil \log p_x \rceil} - 1)$$

The cost for the *cyclic* case is

$$\begin{aligned} T_{rfoldaC} &= t_{leaf} + tyc + txc \\ \text{where} & \\ t_{leaf} &= \left(\frac{n}{p_x p_y} - 1\right) t_e \end{aligned}$$

$$t_{yc} = \begin{cases} \lceil \log p_y \rceil (t_e \frac{n}{p_x p_y} + 2T_{hand} + T_c M \frac{n}{p_x p_y}) + T_c(2^{\lceil \log p_y \rceil} - 1) & \text{if } p_x \text{ is even} \\ \lceil \log(p_y - 1) \rceil (t_e \frac{n}{p_x p_y} + 2T_{hand} + T_c M \frac{n}{p_x p_y}) + T_c(2^{\lceil \log(p_y-1) \rceil} - 1) + & \\ t_{extra_C} & \text{if } p_x \text{ is odd} \end{cases}$$

$$t_{extra_C} = (t_e \frac{n}{p_x p_y} + 2T_{hand} + T_c(M \frac{n}{p_x p_y} + 1))$$

and the cost of the second phase is:

$$t_{xc} = \lceil \log p_x \rceil (t_e \frac{n}{p_x p_y} + 2T_{hand} + T_c M \frac{n}{p_x p_y}) + T_c(2^{\lceil \log p_x \rceil} - 1)$$

Performance models for non-associative folds were also given for blocked and cyclic distributions. The performance model for a block distributed non-associative **rfold** is:

$$T_{rfoldrB} = (p_x p_y) \left(t_e \frac{n}{p_x p_y} + T_c(1+M) + 2T_{hand} \right)$$

And for a cyclic distributed non-associative **rfold** is:

$$T_{rfoldrC} = n \left(t_e + T_c(1+M) + 2T_{hand} \right)$$

No performance models were reported for pipelined *rlist* and *array* primitives. Nonetheless, a performance model for the **iterateFor** skeleton was developed. However, two performance models for pipelining the **iterateFor** skeleton were developed: 1) with stalls 2) and without stalls. For the first case

$$t_{p1} = i(t_{fill} + (n-1)(t_{service} + t_{comms}))$$

and the second

$$\begin{aligned} t_{p2} &= t_{fill} + (i\,n - 1)(t_{service} + t_{comms}) \\ \text{where} & \\ t_{fill} &= p(t_{service} + t_{comms}) \end{aligned}$$

i is the number of iterations, t_{fill} is the time to fill the pipe, n is the number of elements, p is the number of PEs, $t_{service}$ is the service time of a stage, t_{comms} is the communication overhead for a stage and a task.

The first case represents a default pipeline where at the end of each iteration a stall is enforced. This means that the process at the end of the pipe will wait until all processed elements have arrived then it will pipe them into the beginning of the pipe for the next iteration. In other words, for i iterations there will be i independent pipelines.

The second case, which is better, is to interleave the feedback stream of elements with the incoming initial list of data elements rather than stalling. In this case, the cost for filling the pipe is paid once rather than for each iteration. However, to achieve this behaviour for the pipeline it is required that the computation of each element is independent of its neighbours which may not always be possible.

Developer	Type of Model	Verified	Combined
Bratvold	measurement analysis	Yes	No
Busvine	measurement analysis	Yes	No
Campbell	asymptotic analysis	No	No
Cole	asymptotic analysis	No	No
Darlington et al	measurement analysis	Yes	No
Deldarie et al	asymptotic analysis	No	No
Feldcamp and **Sreekantaswamy**	measurement analysis	Yes	No
Skillicorn and **Cai**	measurement analysis	No	Yes
Sreekantaswamy	measurement analysis	Yes	No
Rangaswami	measurement analysis	Yes	Yes
To	measurement analysis	No	Yes

Table 5: A comparison between the cost models

5 A Comparison between the Cost Models

We are interested in the techniques used for modelling algorithmic skeletons. Here, we try to give a general comparison between all the cost models that were developed for algorithmic skeletons. Table 5 presents such a comparison. The criteria used for comparing the cost models are based on the following:

- Type of cost model: which would be either asymptotic- or measurement-based cost model.
- The verification of the cost model against a parallel machine.
- Has the cost model been combined with other cost models.

6 Related Work

There is lots of work on performance modelling of parallel programs. *Jonkers* [14] has presented a survey of of probabilistic performance modelling techniques and their applicability to parallel applications. He outlined different modelling techniques: queuing, task graph, petri net and hybrid models. *Wagner* [22] has reviewed current approaches to modelling (both analytical modeling based on system parameters, and structural modeling based on the structure of the program. Also, he proposed a combination of these two approaches in a new technique called Lost Cycles Analysis (LCA).

7 Summary

There were two main techniques for modelling the performance of algorithmic skeletons: 1) asymptotic analysis and 2) models that were derived through practical experiments and analytical analysis. The first technique was mainly to have a theoretical study about the skeleton (which need not to have a real implementation). The second one (as its name suggests) was to have a prediction of the execution time for an implemented skeleton.

References

[1] Richard Bird. A Calculus of Functions for Program Derivation. Technical report, Oxford University Computing Laboratory, Programming Research Group.

[2] T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[3] D. J. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, April 1993.

[4] D. K. G. Campbell. *CLUMPS: a candidate model of efficient, general purpose parallel computation.* PhD thesis, University of Exeter, October 1994.

[5] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, London, 1989.

[6] A. Y. Grama et al. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August 1993.

[7] B. Bacci et al. P(3)L - A Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):613–641, 1995.

[8] D. Feldcamp et al. Towards a Skeleton-based Parallel Programming Environment. In A. Veronis and Y. Paker, editors, *Transputer Research and Applications*, volume 5, pages 104–115. IOS Press, 1992.

[9] J. Darlington et al. Parallel Programming using Skeleton Functions. In M Reeve A Bode and G Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe, Munich, Germany*, pages 146–160, Leeds, 1993. Springer-Verlag, LNCS 694.

[10] J. Darlington et al. Structured Parallel Programming. In *Proceedings of Massively Parallel Programming Models Conference*, pages 160–169, Berlin, September 1993. IEEE Computer Society Press.

[11] M. Daneluto et al. The P^3L Language: An Introduction. Technical Report HPL-PSC-91-29, Hewlet-Packard Laboratories, Pisa Science Center, Pisa, Italy, December 1991.

[12] D. Feldcamp and A. Wagner. Parsec - A Software Development Environment for Performance Oriented Parallel Programming. In S. Atkins and A. S. Wagner, editors, *Transputer Research and Applications*, volume 6, pages 247–262. IOS Press, 1993.

[13] J. R. Davy H. Deldarie and P. M. Dew. The Performance of Parallel Algorithmic Skeletons. Technical Report 95.6, University of Leeds, Division of Computer Science, March 1995.

[14] H. Jonkers. Introduction to Probabilistic Performance Modelling of Parallel Applications. Technical Report 1-68340-44(1993)04, Delft University of Technology, Faculty of Engineering, Laboratory of Computer Architecture and Digital Techniques, April 1993.

[15] S. Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, Università di Pisa-Genova-Udine, March 1993.

[16] F. A. Rabhi. Exploiting Prallelism in Functional Languages: a Paradigm -Oriented Approach. In *Abstract Machine Models for Highly Parallel Computers*, April 1993.

[17] R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model.* PhD thesis, University of Edinburgh, 1995.

[18] D. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[19] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Programming*, 28(1):65–84, 1994.

[20] H. V. Sreekantaswamy. Performance Prediction Modelling of Multicomputers. Technical Report 91-27, University of British Columbia, Vancouver, Canada, November 1991.

[21] H. W. To. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, 1995.

[22] Jr. Wagner Meira. Modeling Performance of Parallel Programs. Technical report, The University of Rochester, Computer Science Department, Rochester, New York 14627, June 1995.