Guided Resource Organisation in Heterogeneous Parallel Computing

John Darlington Moustafa Ghanem Yi-ke Guo Hing Wing To

Department of Computing,

Imperial College, London SW7 2BZ, U.K.

E-mail: {jd, mmg, yg, hwt}@doc.ic.ac.uk

Abstract

In this paper we describe how performance models can be used as a more structured approach to the problem of resource allocation when programming heterogeneous parallel systems. Functional skeletons are used in this paper to co-ordinate parallel computation in a heterogeneous system. An advantage of this approach is the ability to associate performance models with the implementations of a skeleton. We describe how these performance models can be used to predict the cost of of a particular resource allocation strategy over an entire program expressed using skeletons. Using a parallel conjugate gradient algorithm as a case study we investigate the approach on a mixed vector and scalar parallel machine when exploiting SPMD and simple MPMD parallelism in the algorithm.

1 Introduction

Heterogeneous structures are becoming an important aspect of high performance computing. Examples of these structures arise from clustering together pools of non-uniform workstations and from using parallel computers with mixed processing units such as scalar and vector units. A fun-

damental difficulty in effectively exploiting these systems in the difficulty of resource allocation. Owing to the different performance characteristics of the processing units and their interconnections the diversity of different ways in which an application's task and data can be partitioned, scheduled and mapped onto the available processors is far greater in these types of systems than in homogeneous systems.

The traditional approach to programming such systems is through the use of a sequential language, such as Fortran, coupled with a standard message passing library, such as MPI [11]. Unfortunately, in this approach it is difficult to predict the cost of a particular resource allocation strategy. Indeed the traditional method for choosing a resource allocation strategy is to perform a process known as "performance debugging", whereby a programmer repeatedly executes a program under different resource strategies. By observing the results of these runs, the programmer adjusts the resource allocation decisions in an attempt to improve the performance of the program. Furthermore, although there is usually a level of portability at the language level, there is a lack of performance portability. When an application is moved from one platform to another, this process of performance debugging must be repeated. This problem is compounded by the difficulty in re-expressing the resource allocation strategy owing to the low-level nature of the resulting programs.

In this paper we propose an alternative approach to this problem based on the concept of programming through a set of pre-defined components known as skeletons [5]. These skeletons abstract application level operations and enables a higher-level more abstract approach to parallel programming. Underlying each skeleton is a set of parallel implementations each suited to different applications of the skeleton. An advantage of this approach is the ability to associate performance models with each of these implementations. These performance models provide quantitative predictions of instantiations of the skeletons and can be used for predicting the cost of a particular resource allocation strategy. Thus it is possible to select an appropriate resource allocation strategy based on these predictions.

We begin in Section 2 with an overview of the process behind using performance models to

guide resource allocation in skeleton programs. Section 3 describes the target architecture on which the case study has been conducted. A selection of the primitive skeletons and their performance models is describe in Section 4. Section 5 shows how a suite of application-level skeletons can be built from the skeletons and how their performance models can be derived. A case study implementing an application program, parallel conjugate gradient solver, using the application-level skeleton is described in Section 6. Experimental results demonstrate the accuracy of the models derived for the application program. Section 7 describes how these models can be used as an aid to guide resource allocation.

2 Performance guided resource allocation

This paper investigates how performance models can be used as a more structured approach to the problem of resource allocation when programming heterogeneous parallel systems. This approach is particularly suited to the skeleton methodology as effective performance models can be generated for known program structures. In contrast it is generally more difficult to generate performance models for arbitrary message passing programs. In this section we introduce the main concepts of programming with skeletons, describe the techniques for deriving performance models for skeletons and programs written using skeletons, and discuss how these performance models can be used for guiding resource allocation.

2.1 Parallel programming with skeletons

The philosophy underlying the skeleton approach, is to construct parallel programs from a restricted set of operators known as skeletons. A skeleton is an abstraction of some useful parallel computational structures commonly found in applications. Often a skeleton is a higher-order function, thus it can be specialised towards particular applications. At Imperial College, we have developed an instance of this approach called *Structured Parallel Programming* (SPP(X)) [5]. In the SPP(X) model a *Structured Co-ordination Language* (SCL) is used to co-ordinate the parallel activities of tasks defined in an imperative language, such as Fortran or C. SCL consists of a

collection of skeletons that abstract parallel behaviour such as data distribution and data parallel operations. The skeletons in SCL are defined in a functional language which resembles Haskell [8]. Since the parallel behaviour of a skeleton is known a priori the skeleton can be implemented by pre-defined libraries or code templates in the desired imperative language together with standard message passing libraries providing both efficiency and program portability.

An advantage of the skeleton approach is that as all the skeletons are pure functions, they can be easily composed and therefore modularity and extensibility are naturally supported. Thus it is possible to build up suites of application-level skeletons suitable for building applications. For example, in this paper we explore a number of linear algebra skeletons often used when writing numerical applications.

A full description of the SPP(X) language can be found in [5]. Examples of application-level skeletons are given in Section 5 and examples of programs written in SPP(X) and the application-level skeletons are given in Section 6. A prototype compiler for the language is currently under development. The results from this paper were obtained from a mixture of automatic and hand compiled code.

2.2 Performance models for SCL skeletons

The structure of an implementation of a skeleton, including its data distribution and pattern of communication, is known. This enables the production of a performance model for the implementation. The performance model associated with a SCL skeleton is a function parameterised by the problem and machine characteristics, and returns a prediction of the total time to execute a particular instantiation of the skeleton. The process of producing and verifying a performance model for a given SCL skeleton and machine pair is performed by hand by the system provider. However, this process need only be performed once and the set of SCL constructs or skeletons is limited. These models can then be used to derive performance models for programs or application-level skeletons written in SPP(X).

The performance model for a SCL skeleton is developed once for each machine by first analysing

its implementation on the machine to develop a performance formula, and then by benchmarking to find the constants of the model. In this approach the effects of all machine characteristics, such as processor speeds and network latencies are captured in the constants of the derived model. With some heterogeneous systems the network latency may be non-deterministic. In such cases the network latency would have to be modelled either with a probabilistic model or a worst case model. The only remaining variables in the model are the problem parameters, such as the performance of the sequential code and the problem size, and the number of processors used.

2.3 Developing models for application-level skeletons and programs

The level of abstraction for programming can be raised by developing application-level skeletons. These can be defined in SPP(X) by combining SCL skeletons to express the parallel co-ordination and by providing the sequential code to be executed on the processors. Performance models for the application-level skeletons can then be derived from the performance models of the SCL skeletons and by developing models for the sequential components.

The component of the performance model arising from using the SCL skeletons can be generated automatically by recursively replacing each occurrence of a SCL skeleton with the performance model for that skeleton. The remaining problem is to generate performance models for the sequential code used to instantiate the SCL skeletons. In this paper we do not consider methods for automatically deriving performance models for sequential code, although this is part of our ongoing research. Instead we propose a system which interactively requests the performance models from the application-level skeleton developer. Thus another advantage of developing application-level skeletons is that this process of generating performance models for the sequential code need only be performed once for the given class of applications.

Performance models of applications written in SPP(X) and using application-level skeletons can be derived automatically from the performance models of the SCL skeletons and application-level skeleton. Note that in this scenario the system will not request from the programmer any performance models for the sequential code as these have already been modelled as part of the

application-level skeletons.

2.4 Resource allocation selection and optimisation

Using the scheme described in Sections 2.2 and 2.3 it is possible to systematically derive a performance prediction formula for a given application in terms of its input parameters, such as the problem size, and the number of processors allocated. Once the final quantitative prediction formula is derived for a program it can then be used to estimate the total time required to execute the program for a given set of input parameter values and a given number of processors. The same formula can also be used to calculate the optimal number of processors that provide the minimum execution time for a program. Given the numerical nature of the models, this can be done using one-variable numerical minimisation algorithms (c.f. Chapter 10 of [12]).

For a given heterogeneous system, different resource usage strategies for an application can be coded as semantically equivalent SCL programs. A separate performance model for each version of the program can be automatically derived and used to calculate the optimal number of processors and minimum execution time for each scheme. This information can be used both by the programmer and/or an automatic resource allocation system to choose the implementation that provides the overall minimum execution time.

2.5 Related work

The use of performance models to guide the resource allocation in skeleton programs has been studied by several authors for homogeneous architectures [2, 3, 4]. In particular Bratvold has developed a system for profiling the sequential code to interactively develop performance models of the sequential code fragments based on sample input data sets. The sequential code fragments are written in a functional language, but the concepts should be transferable to an imperative language.

A closely related piece of work is the ALPSTONE Project by Kuhn and Burkhart [10]. The project comprises of both a methodology and a set of tools for guiding a programmer during the

development of a parallel program. In particular the system provides facilities for the derivation of a time model for predicting a program's performance from a high-level description of the program in terms of skeletons. The derivation of performance models through a combination of benchmarking for basic components and arithmetic manipulation for compositions of components is similar to that used in this paper. However, our work extends the basic concept by showing, through a case study, how the technique can be applied to a larger class of algorithms than just SPMD programs and on heterogeneous architectures.

Balasundaram et al. have developed a related tool for statically estimating the relative efficiency of different data partitioning schemes for Fortran D program [1]. The scheme is based on the notion of benchmarking a number of kernel routines which are the constructs of the language. The initial results from modelling the sequential code are promising. The advantage of the skeleton approach is that the more structured organisation of the parallel behaviour of a program, such as communication, should lead to more accurate models for this aspect of a program.

An alternative approach to tackling the performance issues in parallel programming is proposed by Ferscha and Johnson [6]. Their N-MAP tool enables the performance prediction of programs at various stages of development, from high-level abstract descriptions to completely executable codes. The predicted performance of a program is obtained from a discrete event simulation of the program. Although this approach will capture a more general class of programs, the skeleton approach can avoid the overheads of performing simulations by only using pre-defined components with given performance models. The additional structural information inherent in a skeleton should aid in any automatic resource allocation decisions.

3 A heterogeneous architecture

Before describing the skeletons and their performance models it is necessary to introduce the target architecture. In this study we used the Fujitsu AP1000 located at Imperial College. The basic architecture consists of 128 scalar Sparc processors connected by a two-dimensional torus for

general point-to-point message passing and a dedicated network for broadcasting data [9]. Each scalar processor has a theoretical peak performance of approximately 5.6MFLOP/s. Interestingly, 16 of the scalar nodes have Numerical Computational Accelerators (NCA) attached to them. Each NCA consists of an implementation of Fujitsu's μ-VP vector processor each of which has a theoretical peak performance of 100MFLOP/s [7]. Communication between the two units on a single board is through a dedicated shared 16MB DRAM. This results in a heterogeneous architecture which can be exploited in many different ways.

4 Some SCL skeletons and their performance models

Rather than present a complete syntax for the SCL language, we will only introduce a selection of the SCL skeletons which are used by the case study described in this paper. However, this is a representative set which highlights the different styles of performance models and gives a flavour of the SCL language.

4.1 Data distribution

One of the essential aspects of data parallelism which SPP(X) abstracts is data distribution and alignment. For example given two one-dimensional arrays (vectors) `v1` and `v2`, the following use of the SCL skeleton `distribution`:

```
< dv1, dv2 > = distribution [(block p, id), (block p , id)] [v1, v2]
```

distributes both the vectors blockwise into `p` blocks and aligns the distributed blocks of `v1` with `v2`, i.e. the ith block from `v1` is mapped onto the same processor as the ith block from `v2`. The notation `[...]` represents a list. Note that for each element in the list of vectors to be distributed, there must be a corresponding element in the list of distribution patterns. Each distribution pattern has two elements: the first is the distribution scheme used, the second is a data movement function for expressing initial data movement. In this example, there is no initial data movement hence the second element of the distribution patterns is `id` the identity function. The result of the distribution, referred to as a *configuration*, is `< dv1, dv2 >` which represents the list of aligned

Component	Scalar model (μs)	Vector model (μs)
$t_{brdcast}(N)$	$260 + 1.45N$	$160 + 2.2N$
$t_{gather}(P, N)$	$150P + 0.72N$	$150P + 1.28N$
$t_{fold+}(P)$	$130 + 30\log_2 P$	$175 + 270\log_2 P$

Table 1: Performance models of first-order SCL skeletons.

distributed arrays, where `dv1` is the distributed version of `v1` and `dv2` is the distributed version of `v2`. In this paper the distribution phase is treated as part of the program setup time and is therefore not modelled.

4.2 Skeletons for computation and data-movement

The other SCL skeletons abstract the organisation of parallel computation and the data-movement of distributed data structures. We begin by describing some SCL skeletons which are not higher-order functions. The first is `brdcast` which given a data item and a configuration duplicates a copy of the item to each of the sets of aligned distributed segments. This abstracts the concept of broadcasting a given data item to each processor. The next skeleton `gather` collects together the components of a distributed data structure to reform a sequential data structure. The skeleton `fold (+)` is treated as a primitive skeleton in this paper. This skeleton sums together all the distributed segments of a distributed array using addition. The performance models for these skeletons are summarised in Table 1. The parameter N is the size of the problem and P is the number of processors. There are two models for each skeleton depending on whether the skeleton is implemented on the scalar or vector units. Notice that the cost of the `fold (+)` function for the vector units is higher than for the scalar units. This is as a result of the irregular distribution of the vector units. Interestingly there is a difference between the models even for the skeletons which perform no computation as there is a difference in the cost of transferring data to and from the vector processor's memory.

Interestingly the performance model for `brdcast` does not depend on the number of processors involved as the implementation exploits the dedicated broadcast network of the AP1000. The cost for a `gather` is linear, rather than logarithmic, to the number of processors involved owing to the

underlying implementation of `MPI_Gather`.

A common higher-order skeleton is `map` which takes two arguments, a function and a configuration. It applies the given function to each aligned set of distributed segments. For example, `map f < da, db >` will apply `f` to each pair of aligned segment from `da` and `db`. As there are no parallel overheads the performance model for this can be defined as:

$$t_{map}(t_f) = t_f$$

where t_f is the cost of applying `f` to a local data set. This model assumes that the computation is load balanced.

The `SPMD` skeleton captures the pattern of control flow found in the single program multiple data model. The skeleton takes as arguments a pair of functions and a configuration, e.g. `SPMD (g ,f) <da, db>`. It begins by applying the second of the pair of functions (`f`) to every distributed segment of the configuration, then it applies the first function (`g`) globally across the configuration. Examples of functions which can be applied globally across a configuration are the skeletons `fold (+)` and `gather`. As the two phases of a `SPMD` skeleton occur successively, the performance model for the skeleton is the sum of its components:

$$t_{SPMD}(t_g, t_f) = t_g + t_f$$

where t_f is the cost of applying the second function to a distributed segment of the configuration, and t_g is the cost of applying the first function across the entire configuration.

Similarly the skeleton `MPMD` abstracts the multiple program multiple data model. In the `MPMD` skeleton different parallel computational tasks are applied in parallel over different configurations. This can be defined as:

```
MPMD [f1, f2, ... fn]  [c1, c2, ... cn] = [f1 c1, f2 c2, ... fn cn]
```

The first argument is a list of parallel tasks, whilst the second argument is a list of configurations. MPMD provides a simple means to specify the concurrent execution of independent tasks over

different groups of distributed data objects. The performance of the `MPMD` skeleton is the maximum of the times taken by any of its concurrent tasks plus some overhead for setting up the concurrent tasks. In this particular case there are no overhead cost, therefore for simplicity this cost has been omitted from the model:

$$t_{MPMD}(t_{f1}, \ldots, t_{fn}) = \max(t_{f1}, \ldots, t_{fn})$$

where t_{fi} is the cost of applying `fi` to `ci`.

SCL also provides other skeletons including `iterUntil` whose arguments are a step function, a terminating function, a convergence condition and a configuration, e.g. `iterUntil s t c x`. The skeleton iteratively applies the step function (`s`) to the configuration (`x`) until the condition function (`c`) is satisfied whereupon the terminating function (`t`) is applied to it. The performance model for this is:

$$t_{iterUntil}(t_s, t_t, t_c) = i(t_s + t_c) + t_t$$

where t_s, t_t and t_c are the models for applying the step, terminating and condition functions respectively, and i is the number of iterations.

5 A suite of linear algebra skeletons

To demonstrate the principles behind building performance models for application-level skeletons we shall study some matrix and vector operations which are commonly used in expressing numerical applications. In this study vectors are represented by one-dimensional arrays and matrices are represented by two-dimensional arrays. First we define the application-level skeletons in SPP(X) and then we show how the performance models of the skeletons are derived from their definitions.

5.1 Definition of matrix and vector operations

5.1.1 Inner product

The first skeleton considered computes the inner product of two vectors on `p` processors. Given two vectors distributed blockwise on `p` processors their inner product is defined by:

```
innerProduct < dv1, dv2 >  = SPMD( fold(+), S_innerProduct ) < dv1, dv2 >
```

where `S_innerProduct` is the sequential function in Fortran or C for performing a sequential inner product. For example in C this could be written as:

```
void
innerProduct(FLOATTYPE *vec1, FLOATTYPE *vec2, int m,
             FLOATTYPE *result)
{
  int i;
  *result = 0.0;
  for(i=0; i<m; i++)
    *result += vec1[i]*vec2[i];
}
```

The `innerProduct` begins by performing a local, sequential inner product on each aligned pair of the distributed segments of the two vectors, and then performs a global operation summing together all the results of the local inner products.

5.1.2 Scalar-vector product

The next application-level skeleton we shall define performs a scalar-vector product:

```
scalarVectorProduct < s, dv > = map S_scalarVectorProduct < s, dv >
```

where `S_scalarVectorProduct` is the sequential code for performing a sequential scalar-vector product. It has been assumed that the scalar has been duplicated across all the processors.

5.1.3 Vector add

Another useful operator is a generalisation of vector addition. Given two vector v_1 and v_2, and a scalar value α, the operation computes $v_1 + \alpha v_2$. It is assumed that the scalar has been duplicated across all the processors, for example by using the `brdcast` skeleton. The definition of this function is:

```
vectorAdd < dv1, s, dv2 > = map S_vectorAdd < dv1, s, dv2 >
```

where `S_vectorAdd` is the fragment of sequential code for performing a generalised vector addition on the local segments of the vectors. Again the result is computed by performing a sequential

version of the algorithm on the local distributed segments of the vectors. Notice, however, that in this case `S_vectorAdd` must take three arguments, as the locally aligned data structure consists of the segments from the two vectors and a duplicated scalar.

5.1.4 Matrix-vector product

The last application-level skeleton we shall consider computes the multiplication of a matrix and a vector. The method used for computing a matrix-vector product depends on the distribution patterns of the matrix and the vector. The method we shall consider assumes that the matrix `A` is distributed row-wise as `p` blocks and that the vector `x` has been distributed blockwise, for example by the expression:

```
< dA, dx > = distribution [(row_block p, id), (block p, id)] [A, x]
```

Assuming this distribution pattern, a parallel matrix-vector product can be defined as:

```
matrixVectorProduct <dA,dx> = map S_matrixVectorProduct < dA, gather dx >
```

where `S_matrixVectorProduct` is the sequential code for performing a sequential matrix-vector product. The parallel algorithm chosen for implementing a matrix-vector product begins by duplicating the *entire* argument vector on each of the processors. The result can then be computed locally with the result distributed in the same manner as the original matrix.

5.2 Performance models of the application-level skeletons

In this section we begin by describing in detail how a performance model is generated for `innerProduct`. The same technique is then used to produce models for the other application-level skeletons. The resulting models are summarised in Tables 2 and 3.

Recall the definition of `innerProduct`:

```
innerProduct < dv1, dv2 >  = SPMD( fold(+), S_innerProduct ) < dv1, dv2 >
```

The performance model for `innerProduct` can be described in terms of the the performance model for the `SPMD` skeleton:

$$t_{ip} = t_{SPMD}(t_{fold+}(P), t_{sip}(N, P)) \quad (1)$$

Skeleton	Performance model
`innerProduct`	$t_{ip} = t_{sip}(N/P) + t_{fold+}(P)$
`matrixVectorProduct`	$t_{mvp} = t_{gather}(P,N) + t_{brdcast}(N) + t_{smvp}(N/P,N)$
`vectorAdd`	$t_{va} = t_{sva}(N/P)$
`scalarVectorProduct`	$t_{svp} = t_{ssvp}(N/P)$

Table 2: Performance models of linear algebra skeletons.

where N is the size of the vectors and the P is the number of the processors. Notice that the performance model for `SPMD` is parameterised by the performance models of its two functions arguments, `fold (+)` and `S_innerProduct`.

Substituting model of the `SPMD` skeleton back into Equation 1 we arrive at a performance model for `innerProduct`:

$$t_{ip} = t_{sip}(N/P) + t_{fold+}(P) \tag{2}$$

This model is now stated in terms of basic components. A basic component is a performance model that is not constructed from other performance models. These arise from SCL skeletons, as in the case of `fold (+)`, and from fragments of sequential code, as in the case of `S_innerProduct`. Models for the SCL skeletons have already been given in Section 4.

The method used in this paper for modelling sequential code involved first analysing the implementation to produce a performance formula, and then benchmarking to find the constants of this formula. For each of the sequential components there will be two models, reflecting the choice of executing the component on either the scalar or the vector units. Note that where a fragment of sequential code is needed either scalar or vectorised code is used depending on the target processing unit.

Performance models for the other skeletons, `matrixVectorProduct`, `scalarVectorProduct` and `vectorAdd`, can be derived from their definitions in a similar manner to that shown for `innerProduct`. The resulting performance models are summarised in Table 2. Each of the performance models is parameterised by the size of the vector N and the number of processors P. Where appropriate it is assumed that the matrix is square and distributed row-blockwise.

The basic components which arise from the local computations are `S_innerProduct`, `S_matrixVectorProduct`,

Component	Scalar model (μs)	Vector model (μs)
$t_{sip}(N)$	$1.2 + 1.26N$	$2.1 + 0.028N$
$t_{smv}(M, N)$	$1.2 + M(1.56N + 1.2)$	$5.9 + 0.028(M * N)$
$t_{sva}(N)$	$1.2 + 0.56N$	$2.1 + 0.022N$
$t_{ssvp}(N)$	$1.2 + 0.89N$	$2.1 + 0.022N$

Table 3: Performance models of the sequential components.

`S_vectorAdd` and `S_scalarVectorProduct`, with corresponding performance models t_{sip}, t_{smvp}, t_{sva} and t_{ssvp}. For `matrixVectorProduct` the vector must be gathered and broadcasted before the `map` operator is applied. Since this composition of skeletons occurs sequentially, its cost is the sum of the costs of its components. The performance models of the sequential basic components are summarised in Table 3. These are average figures, and do not involve extensive modelling of the cache which would give more accurate models.

6 Parallel Conjugate Gradient Solver

In this section we illustrate, through an example, the method used to automatically generate performance models for application programs from the performance models of skeletons. The case study used is the Conjugate Gradient (CG) method for solving systems of linear equations. In particular we focus on the CG algorithm described by Quinn [13]. Pseudo code for the CG algorithm for solving the system $Ax = b$ is given below:

$k = 0;\ d_0 = 0; x_0 = 0;\ g_0 = -b;\ \alpha_0 = \beta_0 = g_0^T g_0;$
while $\beta_k > \epsilon$ **do**
    $k = k + 1;$
    $d_k = -g_{k-1} + (\beta_{k-1}/\alpha_{k-1})d_{k-1};$
    $\rho_k = d_k^T g_{k-1};$
    $w_k = Ad_k;$
    $\gamma_k = d_k^T w_k;$
    $x_k = x_{k-1} - (\rho_k/\gamma_k)d_k;$
    $\alpha_k = g_{k-1}^T g_{k-1};$
    $g_k = Ax_k - b;$
    $\beta_k = g_k^T g_k;$
endwhile;
$x = x_k;$

where vectors are represented using roman letters (except k) and scalars are represented using

greek letters. We are primarily concerned with the problems of co-ordinating parallel computation and therefore simplify the program by applying it to dense matrices rather than the sparse systems to which it is usually applied. The effect of using sparse rather than dense matrix-vector operations would be to alter the performance model of the individual operations. However, the same technique for constructing an overall performance model for the program and the mechanisms for co-ordinating the operations would remain the same.

The CG algorithm can by expressed in terms of skeletons, by using the skeleton versions of inner product, matrix-vector product and vector addition described in Section 5. Since there is an implementation of each of these skeletons on both the scalar and vector units of the AP1000, it is possible to write semantically equivalent versions of the algorithm which exploit the resources differently. The different implementations we shall consider will exploit the scalar units of the AP1000 only, the vector units of the AP1000 only, and some combination of the two. The many different ways of expressing this algorithm lead to a difficult resource decision over how best to exploit the resources whilst minimising the execution time. In this section we demonstrate how a performance model for each version of the algorithm can be generated from the performance models of the application-level skeletons. To demonstrate the accuracy of the models, we show comparisons between the predicted execution with the measured execution time of the programs.

In Section 7 we discuss how the performance models generated can then be used to aid in choosing an appropriate resource allocation.

6.1 Version 1: Scalar Processors Only

The first implementation only exploits the scalar processors of the AP1000. In this implementation all the vectors are distributed blockwise across the scalar processors and the matrix is distributed row-blockwise across the scalar processors. The following SPP(X) program expresses the CG algorithm:

```
CG A b e = iterUntil iterStep finalResult isConverge
                     (ipG0, < zeroVector, zeroVector, negb, ipG0, ipG0 >)
```

```
where
   <dA,db>@SPG = distribution [(row-block nP, id), (block nP, id)] [A, b]
   ipG0@ROOT    = innerProduct < b, b >
   negb         = scalarVectorProduct < -1, db >
   isConverge (beta, < dx, dd, dg, dalpha, dbeta >) = beta < e
   finalResult (beta, < dx, dd, dg, dalpha, dbeta >) = gather dx
   iterStep (beta, < dx, dd, dg, dalpha, dbeta >)
        = (beta', < dx', dd', dg', alpha', beta' >)
   where  negG       = scalarVectorProduct < -1, dg >
          dd'        = vectorAdd < negG, dbeta/dalpha, dd >
          rho@ROOT   = innerProduct < dd', dg >
          w          = matrixVectorProduct < dA, dd' >
          gamma@ROOT = innerProduct < dd', w >
          dx'        = vectorAdd < dx, -(rho/gamma), dd' >
          alpha'@ROOT= innerProduct < dg, dg >
          u          = matrixVectorProduct < dA, dx' >
          dg'        = vectorAdd < u, -1, db >
          beta'@ROOT = innerProduct < dg', dg' >
```

where `zeroVector` is a constant vector of zeros of size `b` distributed in the same manner as `db`, and `nP` returns the number of processors used. The distribution of the data onto the scalar processors is specified by the notation `@SPG` where `SPG` is the scalar parallel group of processors. The result of the inner products is placed on a unique processor specified by `ROOT`.

The performance model for this program can be automatically generated from its definition. This is achieved by iteratively replacing each of the skeletons with its model. Thus an initial model for Version 1 is:

$$t_{cg1} = t_{iterUntil}(t_{iterStep}, t_{finalResult}, t_{isConverge}) \tag{3}$$

where $t_{iterStep}$, $t_{finalResult}$ and $t_{isConverge}$ are the cost of executing the `iterStep`, `finalResult` and `isConverge` functions respectively. By instantiating the model for $t_{iterUntil}$ we have:

$$t_{cg1} = i_{iter}(t_{iterStep} + t_{isConverge}) + t_{finalResult} \tag{4}$$

where i_{iter} is the number of iterations. The cost of computing $t_{iterStep}$ can also be derived from its definition. Table 4 shows the cost of each statement of the `iterStep` function. In general the cost of each statement is the cost of the performance model of the application level skeleton used in that statement. Notice, however, the use of the shorthand notation for a broadcast in computing some values, such as `dd'` where the scalar value `negG` must be broadcasted. Since each statement

Statement	Cost
`negG = scalarVectorProduct < -1, dg >`	t_{svp}
`dd' = vectorAdd < negG, dbeta/dalpha, dd >`	$t_{brdcast} + t_{va}$
`rho@ROOT = innerProduct < dd', dg >`	t_{ip}
`w = matrixVectorProduct < dA, dd' >`	t_{mvp}
`gamma@ROOT = innerProduct < dd', w >`	t_{ip}
`dx' = vectorAdd < dx, -(rho/gamma), dd' >`	$t_{brdcast} + t_{brdcast} + t_{va}$
`alpha'@ROOT= innerProduct < dg, dg >`	t_{ip}
`u = matrixVectorProduct < dA, dx' >`	t_{mvp}
`dg' = vectorAdd < u, -1, db >`	$t_{brdcast} + t_{va}$
`beta'@ROOT = innerProduct < dg', dg' >`	t_{ip}

Table 4: Cost of the statement in `iterStep` in `cg1`.

is executed sequentially the cost of the function is the sum of the cost of the statement, thus:

$$t_{cg1} = i_{iter}(4t_{ip} + 4t_{brdcst} + 2t_{mvp} + 3t_{va} + t_{svp} + t_{isConverge}) + t_{finalResult} \quad (5)$$

Note that $t_{isConverge}$ is a t_{map} of negligible costs and that the cost of $t_{finalResult}$ is t_{gather}.

To investigate the accuracy of this model experiments were conducted which compared the measured performance of the program with the predicted performance. Experiments were conducted for a fixed number of iterations in order to exclude the effect on the number of iterations caused by differences in the accuracy of the arithmetic operations between the scalar and vector units. This enables comparisons to be made between alternative runs and across the two processing units. The reported results are thus standardised at 100 iterations for all experiments and exclude the time required to initially distribute and finally to collect the data.

The results of the experiments are shown in Figure 1. The first graph, 1(a), shows the execution time vs. the number of processors for several problem sizes. The second graph, 1(b), shows the execution time vs. the problem size for several different processors numbers. The plotted dots represent the measured times, whilst the lines are the predicted times. The predictions are within 10% of the measured times and follow the trend of the measured times.

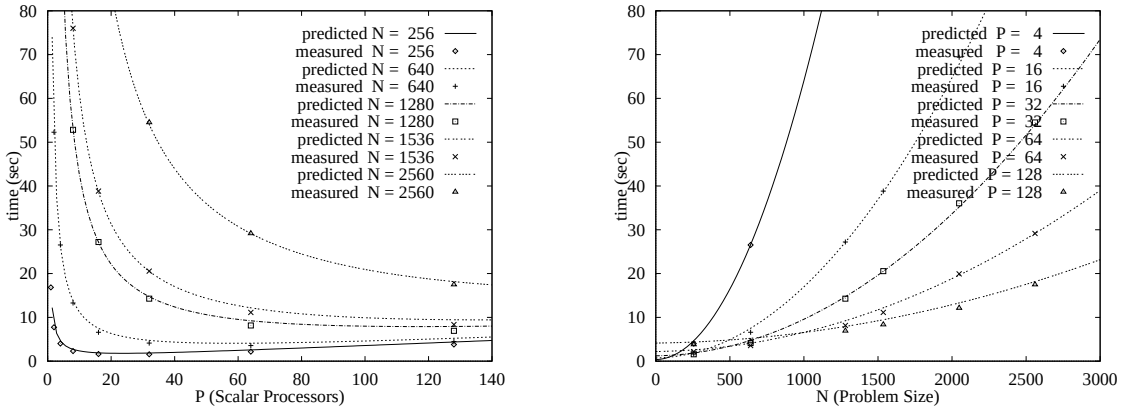

Figure 1: Elapsed time for the parallel scalar case for various processor numbers and problem sizes.

6.2 Version 2: Vector Processors Only

The second implementation only utilises the vector units of the AP1000. Here the data is only distributed to the cells that have vector units attached to them. The difference between this code and the code in Version 1 is the change in the placement of the data from the scalar processor group `SPG` to the vector processor group, `VPG`. Naturally the vector versions of the matrix-vector operations must be used. Owing to the similar nature of the code, the performance model of the program is the same as that of Version 1, but uses the vector versions of the performance models instead of the scalar versions.

Again experiments were conducted to investigate the accuracy of the model. The timing was performed over the main loop for 100 iterations. The results of the experiments are shown in Figure 2. The layout of the graphs is the same as for the experiments for Version 1. The predictions are also within 10% of the measured times and again follow the trend of the measured results.

The results of this experiment reflect the superior performance of the vector units over the scalar units. In general the performance of the program for different processor numbers is as expected. However, notice that for very small problem sizes, a smaller number of vector units performs better owing to the high communication overheads of using more processors.

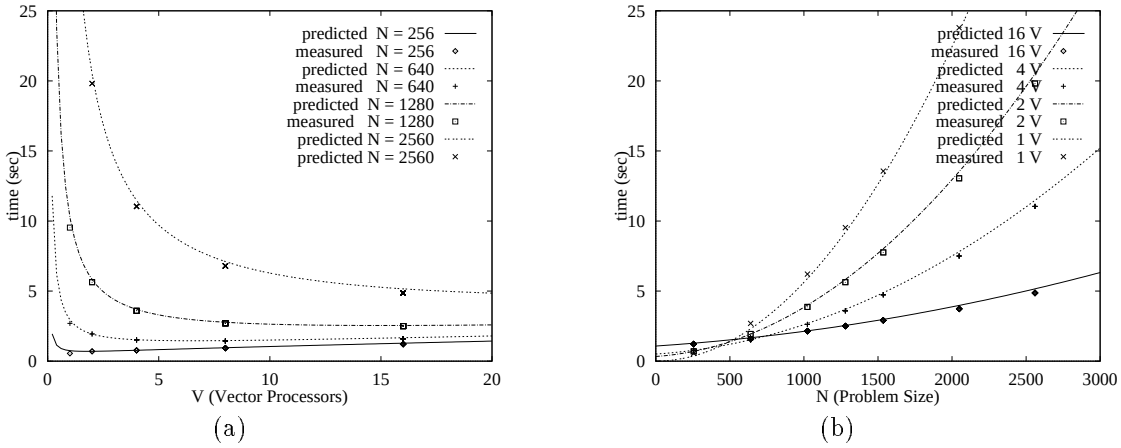

Figure 2: Elapsed time for the parallel vector case for various processor numbers and problem sizes.

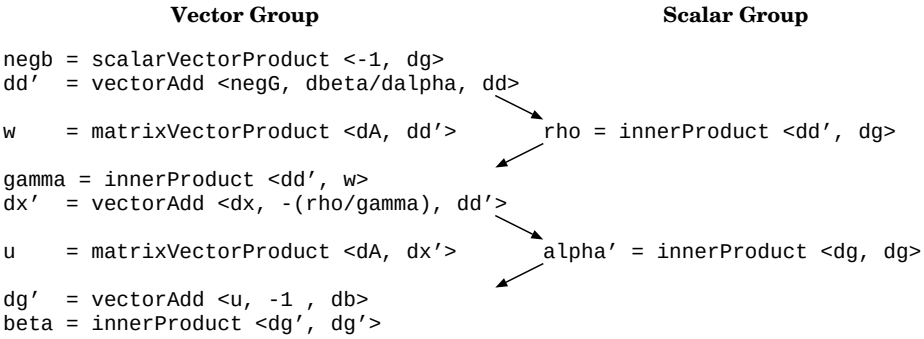

Figure 3: Data dependencies in the CG algorithm.

6.3 Version 3: Mixed Scalar and Vector Processors

The third implementation explores the use of both the scalar and the vector units in this algorithm. A study of the algorithm shows that the computation of the matrix-vector products can be overlapped with some of inner products as there are no data dependencies between these operations. A sketch of the data dependencies in the algorithm is shown in Figure 3. It is thus possible to execute the more expensive matrix-vector product on the vector units whilst concurrently executing inner products on the scalar units. This can be implemented in SPP(X) as:

```
CG A b e = iterUntil iterStep finalResult isConverge
                     (ipG0, < zeroVector, zeroVector, negb, ipG0, ipG0 >)
   where
      <dA,db>@VPG = distribution [(row-block nP, id), (block nP, id)] [A, b]
      ipG0@ROOT    = innerProduct < b, b >
      negb         = scalarVectorProduct < -1, db >
      isConverge (beta, < dx, dd, dg, dalpha, dbeta >) = beta < e
```

```
finalResult (beta, < dx, dd, dg, dalpha, dbeta >) = gather dx
iterStep (beta, < dx, dd, dg, dalpha, dbeta >)
     = (beta', < dx', dd', dg', alpha', beta' >)
where  negG       = scalarVectorProduct < -1, dg >
       dd'        = vectorAdd < negG, dbeta/dalpha, dd >
       [ rho@ROOT, w ] = MPMD [ innerProduct, matrixVectorProduct ]
                              [ < dd', dg >@SPG, < dA, dd' > ]
       gamma@ROOT = innerProduct < dd', w >
       dx'        = vectorAdd < dx, -(rho/gamma), dd' >
       [ alpha'@ROOT, u ] = MPMD [ innerProduct, matrixVectorProduct ]
                                 [ < dg, dg >@SPG, < dA, dx' > ]
       dg'        = vectorAdd < u, -1, db >
       beta'@ROOT = innerProduct < dg', dg' >
```

The overlapping of the vector and scalar processing is expressed using the **MPMD** skeleton. Notice that the vectors used in the inner product must be marked as being mapped to the scalar processor group. This will cause redistribution of data when there are more or fewer scalar processors are being used than vector processors.

Owing to the change in the implementation of the algorithm there will be a corresponding change in the performance model for the program. By analysing the program in the same way as described for Version 1 it is possible to arrive at the following performance model:

$$t_{cg3} = i_{iter}(2t_{ip} + 4t_{brdcst} + 2t_{redist} + t_{MPDM}(2t_{mvp}, 2t_{ip}) + 3t_{va} + t_{svp}) \tag{6}$$

Substituting in the model for t_{MPMD} we have:

$$t_{cg3} = i_{iter}(2t_{ip} + 4t_{brdcst} + 2t_{redist} + \max(2t_{mvp}, 2t_{ip}) + 3t_{va} + t_{svp}) \tag{7}$$

The benchmarked cost of t_{redist} for V vector units, P scalar processors and a problem size N is $(P/V-1)(1.56N+192)\mu s$. Notice that each vector unit only communicates with $(P/V-1)$ other processors rather than P/V processors, since it can use the scalar processor attached to itself as one of the scalar units for performing the overlapped computation.

By comparing the cost of a vectorised matrix-vector product t_{smvp} with the cost of redistribution t_{redist}, it is possible to see that using a different number of scalar processors to vector processor has a prohibitive overhead. When a matching number of scalar processors are used

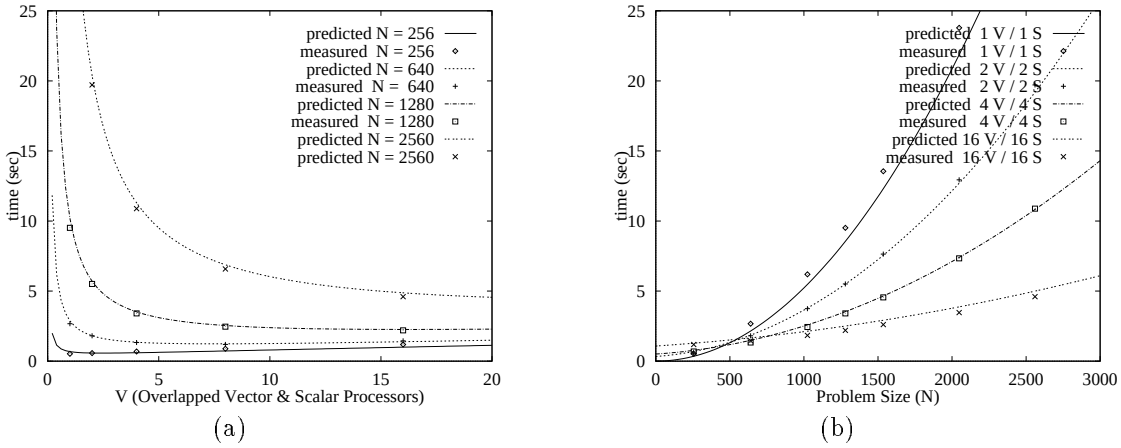

Figure 4: Elapsed time for the mixed parallel vector and scalar case for various processor numbers and problem sizes.

there is no need to redistribute the data as the scalar units attached to the vector units can be used. Therefore the results shown in Figure 4 are for a matching number of vector and scalar processors, where there is no redistribution. If a larger unit of work were being performed by the vector units, there may be a benefit in using a mismatching number of vector and scalar processor as the overhead of redistribution would then be hidden.

The layout of the graphs are as for the previous experiments. The predicted results are within 10% of the measured times and again follow the trend of the measured results. For the majority of the given problem sizes 16 vector and scalar units perform best, although, as in the pure vector case, this is not true for very small problems.

To demonstrate the extra cost of redistribution, Figure 5 shows the execution time vs. the problem size for 16 vector units and 128 scalar units in comparison with 16 vector units and 16 scalar processors.

6.4 Analysis of experimental results

As shown in Figure 6 the best performance was achieved by using a mixture of 16 vector and 16 scalar processors. This combination gives better performance than using only 16 vector processors, or only 128 scalar processors. The performance models predict a similar trend. The experimental results indicates the potential for exploiting heterogeneous environments.

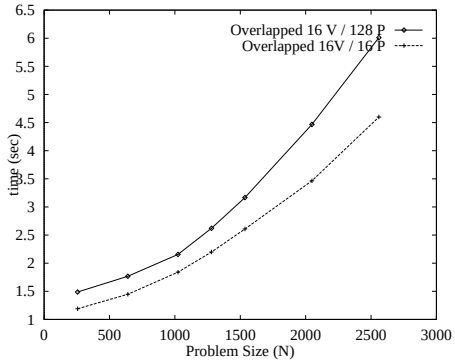

Figure 5: Elapsed time for the mixed parallel vector and scalar case for 16 vector units and 16 scalar processors, and 16 vector units and 128 scalar processors.

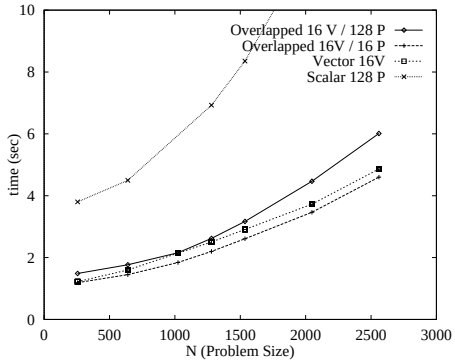

Figure 6: Comparison of execution time for all four experiments vs. problem size.

7 Aiding resources allocation decisions

In this section we briefly describe how performance models can be used as an aid in making resource allocation decisions. Given several semantically equivalent versions of a program, a limited number of resources and a specific problem size, the general problem is to determine the version of a program and the resources to be allocated to it which will achieve the minimum execution time out of the given options. We use as an example the three versions of the CG algorithm described in Section 6. The three versions were scalar non-overlapped, vector non-overlapped and scalar-vector overlapped. For this example the problem size is n the size of the vector, and the resource constraints are p the number of scalar processors available and v the number of vector processors available.

The problem can be solved by first finding for each version of the program the minimum execution time to solve the given problem and the resources required to do so. The solution is then the version of the program which has the minimum predicted time. Note that this process is limited to the accuracy of the performance models. For the case study the performance models were accurate to within 10%. Therefore the predicted number of resources to allocate to a solution is only accurate within 10%. However, this level of accuracy should still be sufficient to differentiate between using different versions of a program and gives a good indication of the number of resources to use.

7.1 Resource allocation for specific program versions

Selecting the minimum execution time from the predictions for each of different versions of the program is straightforward. Thus the principle difficulty in solving the resource allocation problem is to find the minimum predicted execution time for each version of the program. This problem can be expressed as a discrete optimisation problem, where the variables are the resources and the value to be minimised is the predicted execution time. The problem can fall into two classes, one-dimensional problems and multi-dimensional problems.

In the case study the scalar and vector non-overlapped versions are examples of one-dimensional

optimisation problems. The performance model for these programs have two variables, the problem size and the number of processors. The problem size n is fixed, reducing the problem to a single variable. The single variable is the number of scalar processors p (or the number of vector processors v in the vector case). There are a number of existing effective numerical techniques for solving this type of problem (c.f. Chapter 10 of [12]). Note that the range of values for the resource is constrained from 1 to p (or v in the vector case).

The scalar-vector overlapped version is an example of a multi-dimensional optimisation problem. The performance model for this version has three variables, the problem size n, the number of scalar processors p and the number of vector processors v. Since only the problem size is fixed, the formula to be optimised still has two variables. This is a significantly more difficult problem to solve than the one-dimensional problem. The approach adopted in this paper involves an exhaustive search of the problem space, i.e. finding the predicted time for each possible combination of values for p and v. Part of our future work involves comparing this technique with numerical techniques for solving this problem. A more promising direction, however, is to exploit information derivable from the structure of the program to simplify the problem. For example in the scalar-vector overlapped case, the cost of redistribution is significantly higher than the cost of the overlapped computation. To avoid the cost of redistribution the number of scalar processors must be equal to the number of vector processors, i.e. $p = v$. This reduces the problem to a one-dimensional problem. Such techniques could be built into a performance tuning tool as heuristics or the performance models could be presented to the programmer thus allowing the programmer to spot such cases and to place constraints for pruning the search space.

8 Conclusions and Further Work

In this paper we have presented a methodology which uses performance models for guiding resource allocation in programs written for heterogeneous parallel machines. The paper began by describing the techniques involved in developing models for the skeletons and then described how models could

be systematically derived for programs written with skeletons using a case study as an example. The case study reported in this paper, which implemented a parallel conjugate gradient algorithm using different configurations of the vector and scalar processors of an AP1000, has demonstrated the accuracy of the predictions. Methods were described for using the models to aid in determining an appropriate resource allocation strategy from amongst several semantically equivalent versions of a program and for a given problem size. To further validate the approach, further work will include testing the approach on more sophisticated problems and alternative architectures. Of particular interest will be problems and architectures with some degree of non-determinism, such as that caused by load imbalance or network with non-deterministic latencies.

Current work includes completing work on a portable compiler for the SPP(X) system together with an interactive tool for deriving the performance models. Planned future work includes an environment for interactively or automatically generating semantically equivalent, but operationally different programs based on transformations.

Acknowledgements

The authors gratefully acknowledge support from Fujitsu Laboratories, the EPSRC funded project GR/K69988 and the British Council. We would like to thank Fujitsu for providing the facilities at IFPC, which made this work possible.

References

[1] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. *SIGPLAN Notices*, 26(7):213–223, 1991.

[2] Tore Andreas Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Heriot-Watt University, November 1994.

[3] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. In D.B. Skillicorn

and D. Talia, editors, *Programming Languages for Parallel Processing*, pages 205–220. IEEE Computer Society Press, 1994.

[4] J. Darlington, M. Ghanem, and H. W. To. Structured parallel programming. In *Programming Models for Massively Parallel Computers*, pages 160–169. IEEE Computer Society Press, September 1993.

[5] J. Darlington, Y. Guo, H. W. To, and J. Yang. Functional skeletons for parallel coordination. In Seif Haridi, Khayri Ali, and Peter Magnussin, editors, *Euro-Par'95 Parallel Processing*, pages 55–69. Springer-Verlag, August 1995.

[6] A. Ferscha and J. Johnson. N-MAP: A virtual processor discrete event simulation tool for performance prediction in the CAPSE environment. In *Proceedings of the 28th-HICSS Conference*, pages 276–285, Maui, USA, January 1995. IEEE Computer Society Press.

[7] Fujitsu Ltd. *VPU (MB92831) Functional Specifications*, 1.20 edition, March 1992.

[8] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell — a non-strict purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):1–162, 1992.

[9] Hiroaki Ishihata, Takeshi Horie, Satoshi Inano, Toshiyuki Shimizu, Sadayuki Kato, and Morio Ikesaka. Third generation message passing computer AP1000. In *International Symposium on Supercomputing*, pages 46–55, 1991.

[10] Walter Kuhn and Helmer Bukhart. Performance modeling for parallel skeletons: the ALPSTONE project. In *Parallel Programming Environments for High Performance Computing, 2nd European School of Computer Science*, Alpe d'Huez, April 1996. Institut d'Etudes Scientifiques Avancees de Grenoble.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 1.1.* Available from Oak Ridge National Laboratory: `http://www.mcs.anl.gov/mpi`, June 1995.

[12] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992.

[13] Michael J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, second edition, 1994.