Co-ordinating Heterogeneous Parallel Computation

Peter Au John Darlington Moustafa Ghanem
Yi-ke Guo Hing Wing To Jin Yang

Department of Computing,
Imperial College, London SW7 2BZ, U.K.
E-mail: {aktp, jd, mmg, yg, hwt, jy}@doc.ic.ac.uk

Abstract. There is a growing interest in heterogeneous high performance computing environments. These systems are difficult to program owing to the complexity of choosing the appropriate resource allocations and the difficulties in expressing these choices in traditional parallel languages. In this paper we propose that functional skeletons are used to express these resource allocation strategies. By associating performance models with each skeleton it is possible to predict and optimise the performance of different resource allocation strategies, thus providing a tool for guiding the choice of resource allocation. Through a case study of a parallel conjugate gradient algorithm on a mixed vector and scalar parallel machine we demonstrate these features of the SPP(X) approach.

1 Introduction

Parallel computation platforms often now have a heterogeneous structure arising from either exploiting clusters of workstations or from using parallel computers with specialist hardware such as vector units. The recent development of techniques for integrating high speed networks with high performance computers, such as the I-way project [5], provide strong evidence of the growing importance of heterogeneous parallel computing.

To fully exploit the power of a heterogeneous parallel computing system sophisticated resource allocation strategies are often required. This is a complex task as it is difficult to predict the performance of any particular resource allocation strategy on a system composed out of components which have widely differing functionalities and performance. To maximise the overall performance of an application on such systems, a more structured mechanism is needed where the resource requirement can be explicitly specified, and the performance of a resource allocation strategy can be quantified and calculated. This situation often occurs with homogeneous parallel machines, because of the diversities between different machine architectures. However, the problem is exacerbated in a heterogeneous system where the number of possible options is greatly increased and the differences in potential performance more marked.

Traditional approaches to parallel programming, such as the use of lower level communication libraries [7], are often unsatisfactory for this task. The main drawback of these approaches lies with the low-level nature of the mechanisms

used for co-ordinating parallel computation. In these approaches it is difficult to both specify and predict the cost of a particular resource allocation strategy. Furthermore considerable effort is needed to change between resource allocation strategies owing to the low-level nature of the resulting programs.

In this paper, we explore the application of a more structured approach to the problem of programming heterogeneous systems. At Imperial College, we have developed a Structured Parallel Programming language, SPP(X), based on the idea of using a set of higher-order, pre-defined functions, known as *skeletons*, to co-ordinate the parallel activities of tasks defined using standard imperative languages [4]. This high-level approach provides the programmer with powerful control over the allocation of resources, which is necessary for the efficient use of heterogeneous machines. Furthermore this approach to parallel programming provides a uniform and simple method for guiding the complex resource allocation in a heterogeneous system by associating each skeleton with *performance models*. These performance models provide quantitative predictions of the use of the skeleton and can thus be used for predicting the cost of a particular resource allocation strategy.

In this paper, we present a pilot study in using SPP(X) to program a combined vector and scalar parallel computation. In Section 2, we survey the SPP(X) language. In Section 3, we describe performance guided resource organisation. Section 4 presents a set of experiments which implement a parallel Conjugate Gradient algorithm using different resource allocation strategies. Conclusions and further work are described in Section 5.

2 Parallel Programming with Structured Co-ordination

In this section we briefly outline the SPP(X) language. A more detailed description of the language can be found in [4]. In the SPP(X) model a Structured Co-ordination Language, SCL, is used to co-ordinate fragments of sequential code which are written in a base sequential language (X) such as Fortran and C. An application is therefore constructed in two layers: a higher co-ordination level and a lower base language level.

2.1 SCL: A Structured Co-ordination Language

SCL consists of skeletons (or co-ordination forms) that abstract parallel behaviour. Being functions, the skeletons can be easily composed and therefore, modularity and extensibility is naturally supported. Since all parallel behaviour arises from the known behaviour of these skeletons, they 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. SCL has three main sets of constructs: configuration skeletons, data parallel skeletons and computational skeletons.

Configurations and Configuration Skeletons A *configuration* models the logical division and distribution of data objects. To program configurations a set of skeletons known as *configuration skeletons* are provided. For example the `distribution` skeleton defines the configuration of two arrays `A` and `B`:

```
distribution (f,p) (g,q) A B = align (p o partition f A)(q o partition g B)
```

where `o` is functional composition. The `partition` skeleton divides the initial arrays `A` and `B` into parallel arrays of sequential sub-arrays. The two functions `f` and `g` are the partitioning strategies used to divide the arrays. In SCL, some commonly occurring partitioning strategies, such as `row_block`, are provided as built-in functions. The functions `p` and `q` are data-movement skeletons which are described in more detail below. The `align` skeleton pairs the corresponding sub-arrays of two distributed arrays together. The `distribution` skeleton is usually generalised to a list of arrays rather than just two arrays.

The result of applying `distribution` is a parallel array of tuples (a configuration). Each element of the parallel array is a tuple of the sub-arrays that have been allocated to the same processor. As a short hand, rather than writing a configuration as an array of tuples, we can also regard a configuration as a tuple of (distributed and aligned) arrays and write it as $\texttt{<DA}_1, \ldots, \texttt{DA}_n\texttt{>}$ where $\texttt{DA}_j$ represents the distributed version of array $\texttt{A}_j$. In particular we can pattern match to this notation to extract a particular distributed array from a configuration.

It is necessary in some applications to control the placement of configurations onto specific processors. This physical data placement is modelled in SCL by a *processor reference function* `@procSet` which explicitly specifies a mapping from a data configuration to a set of processors defined by `procSet`.

Data Parallel Operators SCL provides a set of skeletons which abstract the basic operations found in the data parallel computation model. For example, the function `map` abstracts the behaviour of executing the same task on all the components of a distributed array. While the function `fold` abstracts a parallel reduction computation over a distributed data structure, such as summing together the components of a distributed array. The skeleton `brdcast` broadcasts a data item over a configuration. Thus `brdcast d C` will associated a copy of the data item `d` with each component of `C`. To simplify the notation for broadcasting, the configuration syntax has been extended. Given a configuration `<dA, dB>` and a value `v` to be broadcast, this can be written as `<dA, dB, v>`.

Computational Skeletons for Abstracting Control Flow Commonly used parallel control flow patterns are abstracted in SCL as *computational skeletons*. For example, the skeleton `SPMD` abstracts the features of Single Program Multiple Data (SPMD) computation. The definition of this skeleton is given as:

```
SPMD [] = id
SPMD (globalFun, localFun):fs = (SPMD fs) o (globalFun o (map localFun))
```

where `h:tl` is a list whose head is `h` and tail is `tl`. `SPMD` takes a list of global-local operation pairs, which are applied over configurations of distributed data objects.

The local operations, `localFun`, are farmed to each processor and computed in parallel. The global operations, `globalFun`, operate over the whole configuration. These global operations are parallel operations that require synchronisation and communication.

Another computational skeleton `MPMD` abstracts the Multiple Program Multiple Data (MPMD) model. This can be defined in a similar way:

```
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. SCL also provides other computational skeletons including `iterUntil s t c x` which iteratively applies `s` to `x` until condition `c` is satisfied whereupon `t` is applied to it.

2.2 Examples of using SCL

To illustrate the basic features of SPP(X) and SCL some short examples are described. Given the configuration of two distributed vectors `dv1` and `dv2`:

```
< dv1, dv2 > = distribution [(block n, id), (block n , id)] [V1, V2]
```

the following program computes the inner product of two vectors by `n` processors:

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

where `S_innerProduct` is the sequential code in Fortran or C for performing a sequential inner product. The function begins by performing a local, sequential inner product on each pair of the distributed segments of the two vectors, and then performs the global operation of summing all the results of the local inner products. Similarly, suppose a matrix `A` is distributed row wise as `n` blocks (e.g. by the expression `dA = partition (row_block n) A`) and a vector `x` has been distributed as for an inner product then 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.

Currently we are building a prototype compiler/translator which translates SPP(X) programs into Fortran or C plus MPI [7] targeted at the AP1000. In this paper, SPP(X) with underlying imperative language as C is used to program the different approaches to the Conjugate Gradient algorithm.

3 Performance Guided Resource Organisation

Organising the computational resources in a heterogeneous parallel machine is often a difficult task owing to the diversity of ways in which an algorithm can exploit these resources. The usual solution is for the programmer to repeatedly execute the program under different resource strategies, observe the results and adjust the resource allocation decisions in an attempt to improve the performance of the program.

An alternative approach is to guide the choice of resource usage by predicting the performance of particular implementation strategies. This approach is very effective when the essential aspects of parallel behaviour are systematically abstracted as known program forms, such as those found in the SPP(X) approach, since each form naturally contains sufficient information for determining the performance under a given resource strategy. Thus, the process of producing and verifying performance models for an SPP(X) skeleton and machine pair need only be performed once. This approach has been adopted by several authors for homogeneous architectures [1, 2, 3].

The performance model associated with a SPP(X) skeleton is usually a function parameterised by the problem and machine characteristics, and returns a prediction of the total time to execute a specific instantiation of a skeleton. A performance model can be developed through a combination of analysing and benchmarking the implementation. In this paper the performance of the primitive SPP(X) skeletons are developed through benchmarking. Techniques for developing performance models through analysis are described in an earlier paper [3].

3.1 A Heterogeneous Architecture

The case study uses 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 [8]. 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 [6]. 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.

3.2 Performance of Matrix and Vector Operations

In this section we highlight the details of performance modelling through several simple examples of performance models for some skeletons and their compositions. These will be later used in the case study described in Section 4. As discussed in Section 2, skeletons can be easily composed to produce higher-level

co-ordination forms. Two examples `innerProduct` and `matrixVectorProduct` have already been defined. Two further higher-level co-ordination forms which will be useful for the case study are:

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

where `S_scalarVectorProduct` is the sequential code for performing a sequential scalar-vector product. The other 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` operator. The definition of the function is:

```
vectorAdd < v1, s, v2 > = map S_vectorAdd < v1, s, v2 >
```

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

From a simple study of the definitions of the four operations, `innerProduct`, `matrixVectorProduct`, `scalarVectorProduct` and `vectorAdd`, performance models based on the performance of the primitive skeletons can be derived:

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

The performance models are parameterised by the size of the vector N and the number of processors used P. It is assumed that the matrix is square and distributed row-blockwise. The components t_{fold+}, t_{gather} and t_{brdcst} reflect the cost of performing the primitive skeletons `fold (+)`, `gather` and `brdcast` (explicitly or using shorthand notation) respectively. Notice that there is no overhead involved in performing a `map` therefore the cost of `vectorAdd` and `scalarVectorProduct` is simply the cost of the local computation. The costs of the local computation performed by each operation are represented by t_{sip}, t_{mvp}, t_{sva} and t_{ssva}.

The performance of implementations of the primitive skeletons and the sequential code fragments used in the matrix-vector operations were benchmarked on the AP1000. For each of the basic components there are 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. The resulting performance models from the benchmarking exercise are:

Component	Scalar model (μs)	Vector model (μs)
$t_{sip}(N)$	$1.2 + 1.26N$	$2.1 + 0.028N$
$t_{smv}(M,N)$	$1.2 + M(1.56N + 1.2)$	$5.9 + 0.0028(M * N)$
$t_{sva}(N)$	$1.2 + 0.56N$	$2.1 + 0.022N$
$t_{ssvp}(N)$	$1.2 + 0.89N$	$2.1 + 0.022N$
$t_{fold+}(P)$	$130 + 30\log_2 P$	$175 + 270\log_2 P$
$t_{gather}(P,N)$	$150P + 0.72N$	$150P + 1.28N$
$t_{brdcst}(N)$	$260 + 1.45N$	$160 + 2.2N$

These are average figures, and do not involve extensive modelling of the cache which would give more accurate models. 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. The lower bandwidth seen by the vector processors for the `gather` and `broadcast` operations is due to cost of transferring data to and from the vector processor's memory.

4 Case Study: Parallel Conjugate Gradient Solver

The SPP(X) approach to co-ordinating heterogeneous parallel computation is illustrated through a case study of the Conjugate Gradient (CG) method for solving systems of linear equations. In particular we focus on the CG algorithm described by Quinn [9]. 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. 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**
    $k = k + 1;$
    $d_k = -g_{k-1} + (\beta_{k-1}/\alpha_{k-1}) d_{k-1};$
    $\rho_k = d_k^T g_{k-1};$
    $w_k = A d_k;$
    $\gamma_k = d_k^T w_k;$
    $x_k = x_{k-1} - (\rho_k/\gamma_k) d_k;$
    $\alpha_k = g_{k-1}^T g_{k-1};$
    $g_k = A x_k - b;$
    $\beta_k = g_k^T g_k;$
endwhile;
$x = x_k;$

where vectors are represented using roman letters (except k) and scalars are represented using greek letters. This algorithm can be parallelised by using parallel versions for the inner products, matrix-vector products and vector additions. Parallel versions of these operations written in SCL were described in Section 3. Since each of these operations can be executed on either the scalar or the vector units, the implementation of the algorithm can exploit only the scalar units of the AP1000, only the vector units of the AP1000, or some combination of the two. This leads to a difficult decision over how to use the resources effectively. The following experiments demonstrate how the performance models developed in Section 3 can be used to accurately predict the cost of various implementations of this algorithm in SCL. It is thus possible to use the performance models to aid in making the appropriate resource allocation. The experiments also emphasise the ease with which these complex resource strategies can be expressed using the SPP(X) approach.

4.1 Experiment 1: Scalar Processors Only

The first implementation only exploits the scalar processors of the AP1000. In this implementation all the vectors are distributed block-wise across the scalar processors and the matrix is distributed row-block-wise 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`.

From the structure of the program it is possible to derive the following performance model for the main loop of the program:

$$t_{cg1} = i_{iter}(4t_{ip} + 4t_{brdcst} + 2t_{mvp} + 3t_{va} + t_{svp})$$

where i_{iter} is the number of iterations. All the experiments are 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 Fig. 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.

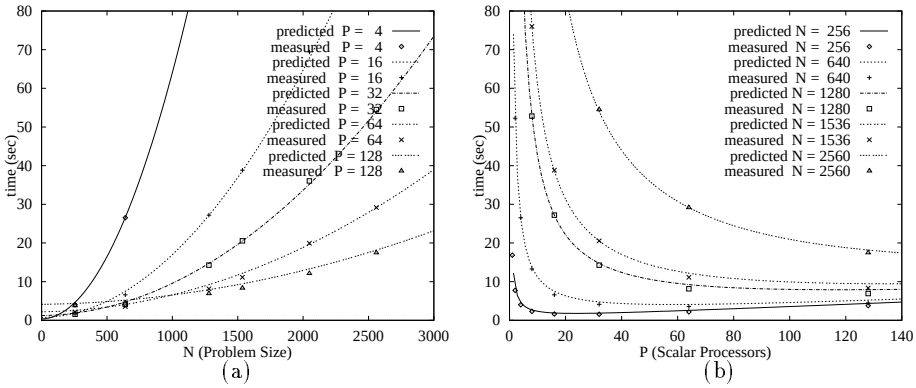

Fig. 1. Elapsed time for the parallel scalar case for various processor numbers and problem sizes.

4.2 Experiment 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 SPP(X) program for this is:

```
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   = 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' >
```

The difference between this code and the code in experiment 1 is the change in the placement of the data from the scalar processor group 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 experiment 1, with but different costs associated with the components. Again the timing was performed over the main loop for 100 iterations.

The results of the experiments are shown in Fig. 2. The layout of the graphs

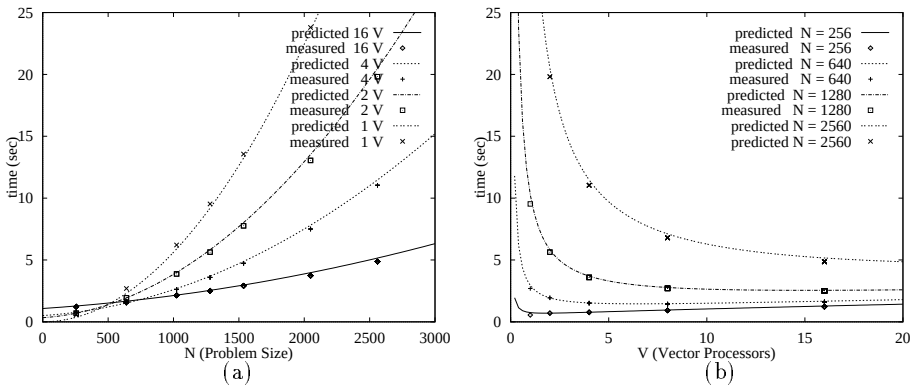

Fig. 2. Elapsed time for the parallel vector case for various processor numbers and problem sizes.

is the same as for experiment 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.

4.3 Experiment 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 Fig. 3. It is thus possible to execute the

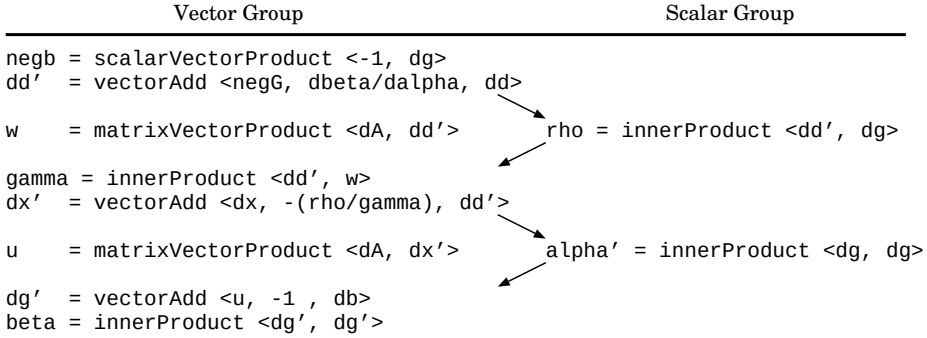

Fig. 3. Data dependencies in the CG algorithm.

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. In cases where more or fewer scalar processors are being used than vector processors, this re-allocation of data will have an associated cost for redistributing the data.

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 it is possible to arrive at the following performance model:

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

where the performance for the `MPMD` skeleton is the maximum of time 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.

$$t_{mpmd}(t_1, t_2) = \max(t_1, t_2)$$

The benchmarked cost of t_{redist} for V vector units, P scalar processors and a problem size N is $(P/V - 1)(3.12N + 192)\mu s$. 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 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 Fig. 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.

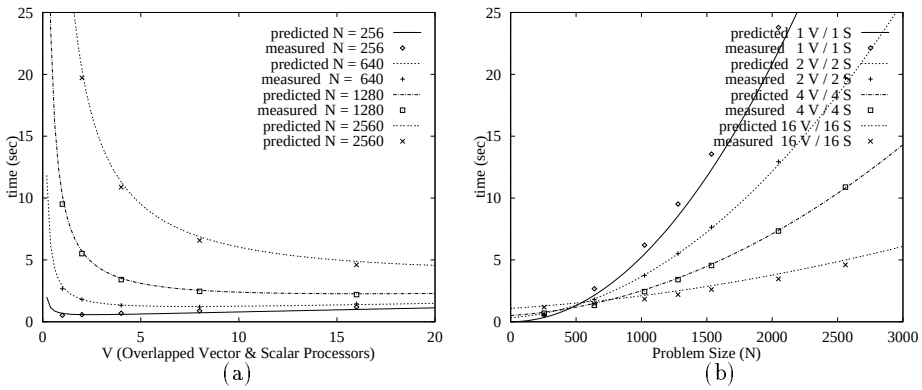

Fig. 4. Elapsed time for the mixed parallel vector and scalar case for various processor numbers and problem sizes.

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, Fig. 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.

4.4 Analysis of Results

As shown in Fig. 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 ability to predict the performance of programs enables the appropriate resource decision to be made without recourse to implementing all the different strategies and executing them. This pilot study indicates the potential for exploiting heterogeneous environments. By using higher-level co-ordination forms different configurations of the machine can be easily organised and their performances systematically predicted.

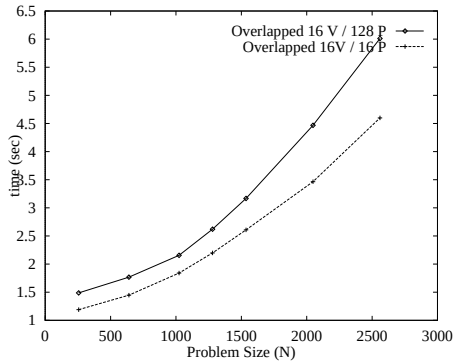

Fig. 5. Elapsed time for the mixed parallel vector and scalar case for 16 vector units and 16 scalar processors, and 16 vector units and 128 scalar processors.

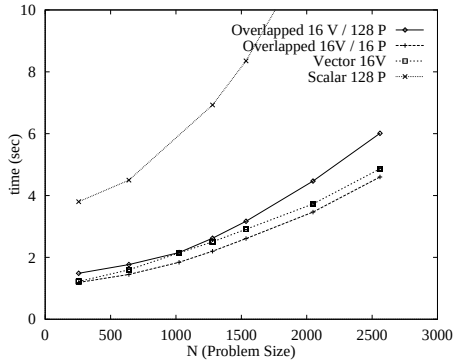

Fig. 6. Comparison of execution time for all four experiments vs. problem size.

5 Conclusions and Further Work

In this paper we have presented a methodology for co-ordinating and organising resources in heterogeneous parallel machines. By using higher-level co-ordination forms, different configuration structures of the machine can easily be expressed and their performances can be systematically predicted. The pilot 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 these features in the SPP(X) approach. The experiments indicate that the structured method of programming by using co-ordination forms allows sufficient control over the allocation of resources to make efficient use of heterogeneous machines. To further validate the approach, further work includes testing the approach on more sophisticated problems such as a multigrid solver and a parallel climate model where the overlapping of the vector computation and the scalar computation will become even more crucial to the overall performance.

The models presented in this paper were generated by hand. However, an analysis of these models will show that the models presented can be derived from the syntax of the program. Current work includes the development of a system for

automatically deriving these models as functions over the basic skeletons. We are currently completing work on a portable compiler for the SPP(X) system together with an interactive tool for performance modelling.

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. Tore Andreas Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Heriot-Watt University, November 1994.
2. 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.
3. 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.
4. J. Darlington, Y. Guo, H. W. To, and J. Yang. Functional skeletons for parallel coordination. In Seif Haridi, Khayri Ali, and Peter Magnussin, editors, *EUROPAR'95 Parallel Processing*, pages 55–69. Springer-Verlag, August 1995.
5. Ian Foster and Warren Smith. I-WAY application developers and users guide version 1.2. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.
6. Fujitsu Ltd. *VPU (MB92831) Functional Specifications*, 1.20 edition, March 1992.
7. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.
8. 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.
9. Michael J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, second edition, 1994.