Co-ordinating Heterogeneous Parallel Computation

Hing Wing To

P. Au J. Darlington M. Ghanem

Y. Guo J. Yang

Department of Computing

Fujitsu Parallel Computing Centre

Imperial College

180 Queen's Gate, London, SW7 2BZ

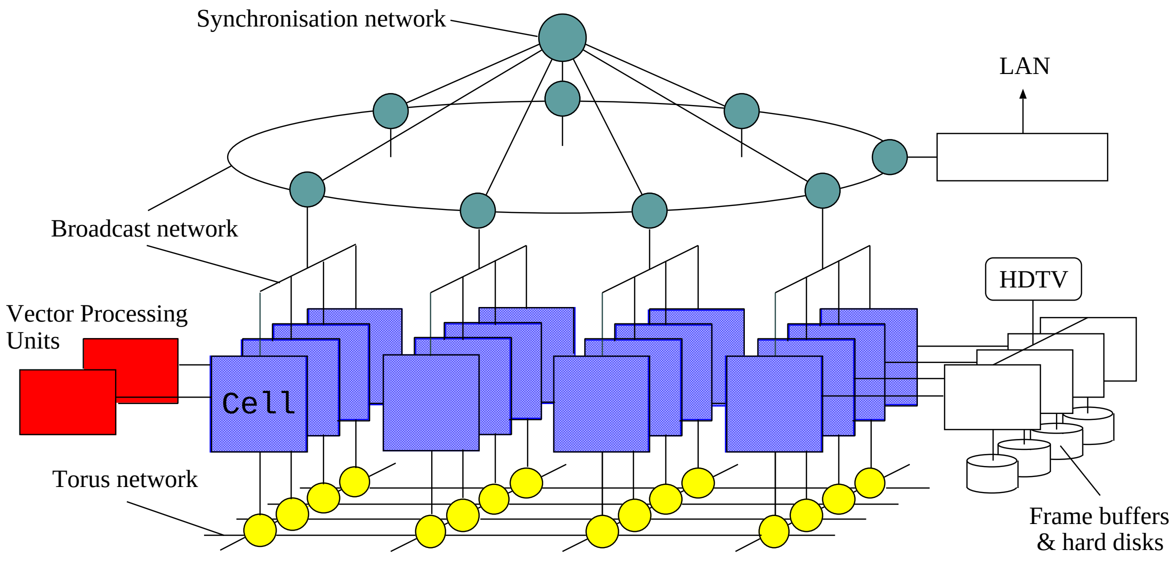
hwt@doc.ic.ac.uk

Heterogeneous Systems

Heterogeneous environments:

- Clusters of workstations.
- Parallel computers with specialist nodes.
- High performance computers connected across networks (e.g. I-Way Project).

For example the Fujitsu AP1000 at IFPC:



Exploiting Heterogeneous Systems

The challenges in programming heterogeneous systems are:

- Expressing the different possible resource usages.
- Deciding between different resource allocation strategies.

Existing approaches:

- Assume universal architecture.
- Use low-level parallel language and follow performance debugging cycle.

Our approach:

- High-level structured language for expressing resource decisions.
- Performance models for guiding the choice of resource strategy.

Structured Parallel Programming

SPP(X) a structured language for co-ordinating the concurrent activities of Fortran code.

A parallel program has two levels:

- SCL = parallel behaviour.
- Fortran = low-level sequential computation.

Structured Co-ordination Language

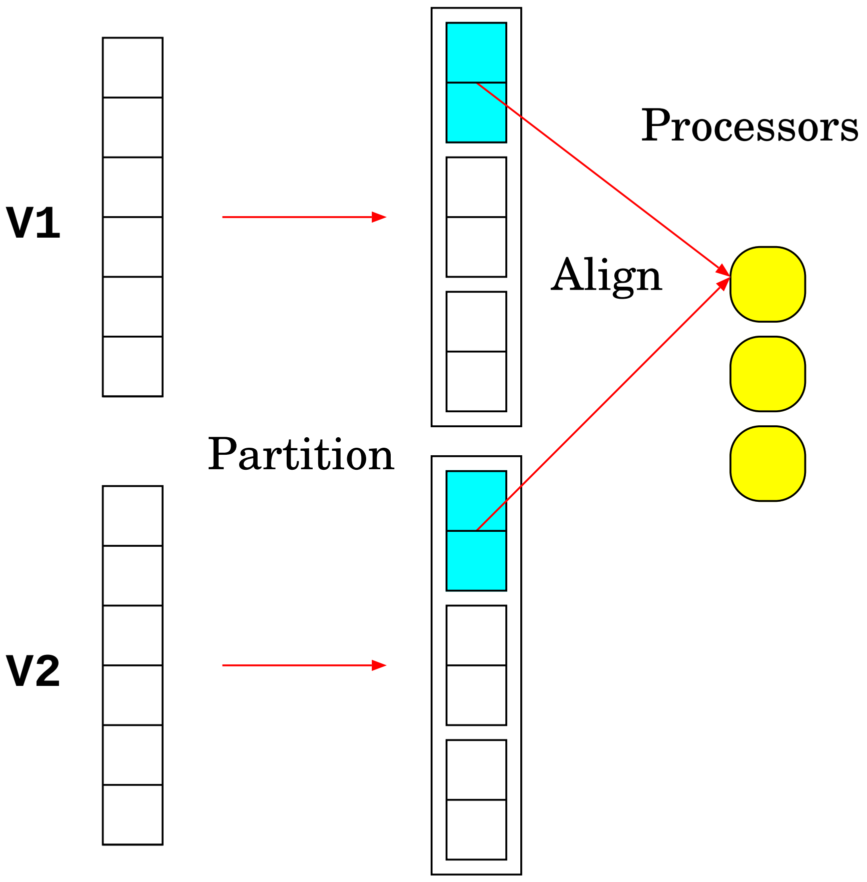
SCL consists of three types of operators (known as *skeletons*):

- Distribution skeletons.
- Computation and communication skeletons.
- Control flow skeletons.

An Example: Inner-Product

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

innerProduct < dA, dB > =
  SPMD( fold(+), S_innerProduct) < dA, dB >
```



Some Matrix-Vector Operations

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


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


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

Performance Models

Models of SCL skeletons by benchmarking:

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

Derive models for application-level skeletons:

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

Sequential code fragments benchmarked:

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

Parallel Conjugate Gradient Solver

The Conjugate Gradient (CG) method is used for solving systems of linear equations:

$$A\,x = b$$

Pseudo code:

```
$k = 0;\ d_0 = 0; x_0 = 0;\ g_0 = -b;\ \alpha_0 = \beta_0 = g_0^T g_0;$
**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**;
$x = x_k;$
```

Can be parallelised by using the skeleton versions of the matrix-vector operations.

Can the operations also be executed in parallel?

Data Dependencies in the CG Algorithm

$$\begin{tikzcd}
d_k = -g_{k-1} + (\beta_{k-1}/\alpha_{k-1})d_{k-1}; \arrow[d] \arrow[dr] & \\
w_k = Ad_k; \arrow[d] & \rho_k = d_k^T g_{k-1}; \arrow[dl] \\
\gamma_k = d_k^T w_k; & \\
x_k = x_{k-1} - (\rho_k/\gamma_k)d_k; \arrow[d] \arrow[dr] & \\
g_k = Ax_k - b; \arrow[d] & \alpha_k = g_{k-1}^T g_{k-1}; \arrow[dl] \\
\beta_k = g_k^T g_k; &
\end{tikzcd}$$

There are a number of ways to exploit the resources of the AP1000:

1. Scalar processors only and execute the statements one after the other.
2. Vector processors only and execute the statements one after the other.
3. Mixed vector and scalar processors, overlap the execution of statments.

Version 1: Scalar Processors Only

```
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' >
```

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

Version 2: Vector Processors Only

```
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)]

      ⋮
```

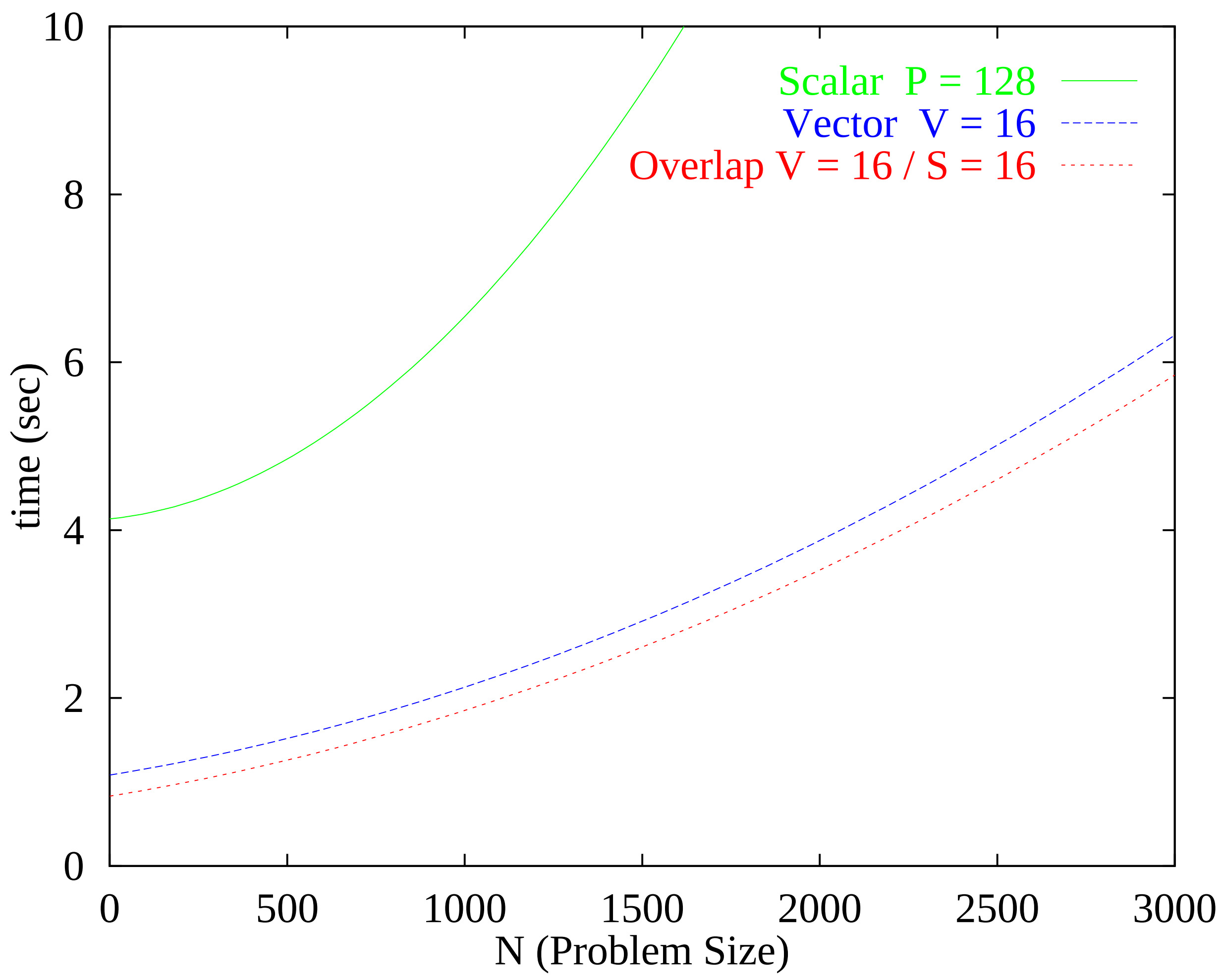
$$t_{cg2} = i_{iter}(4t_{ip} + 4t_{brdcst} + 2t_{mvp} + 3t_{va} + t_{svp})$$

Version 3: Mixed Scalar and Vector Processors

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

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

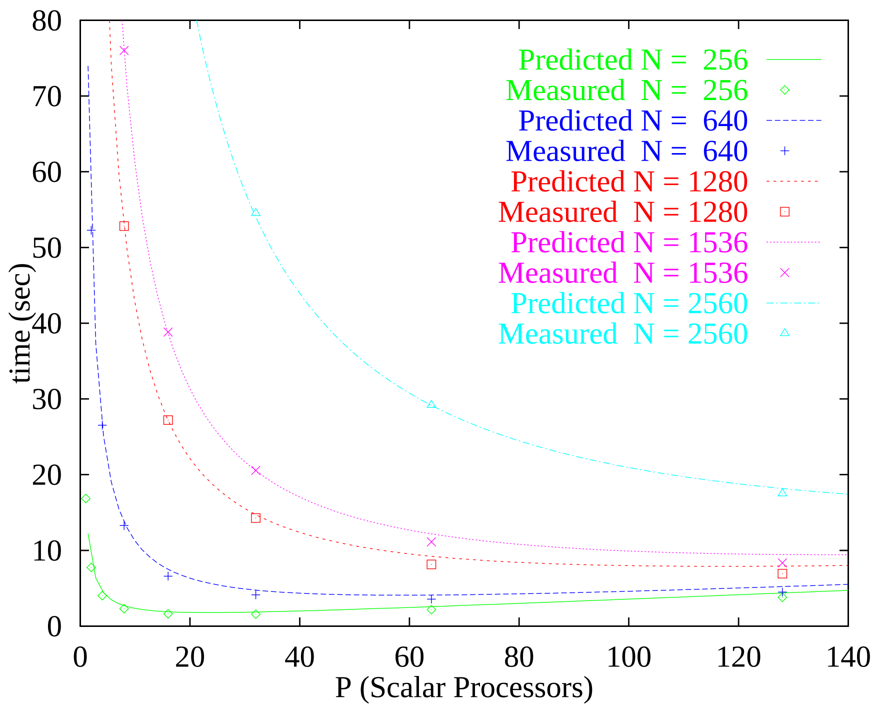
Predicted Performance of the Different Programs

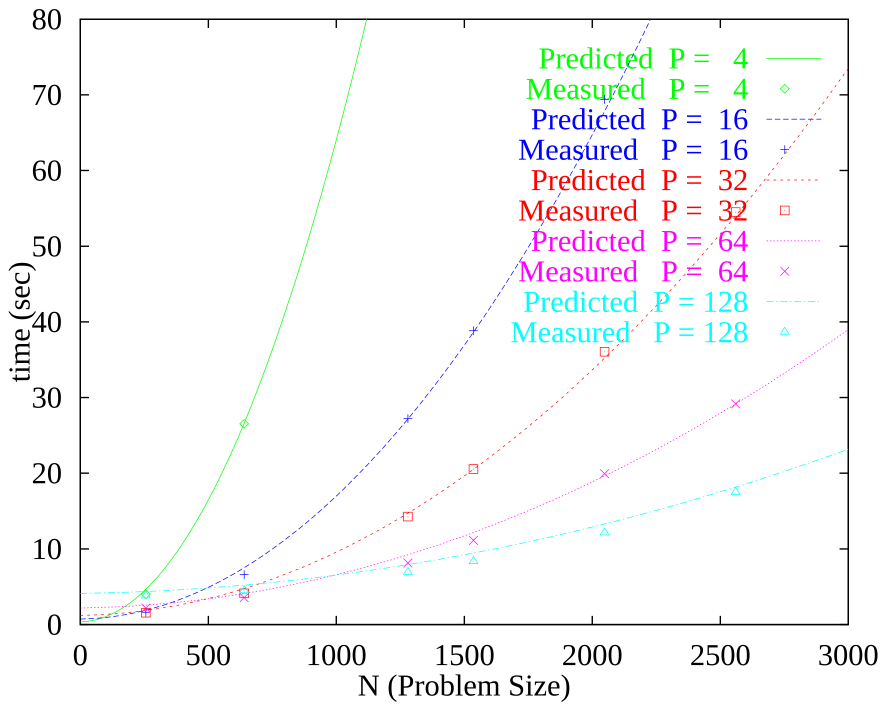


Predict:

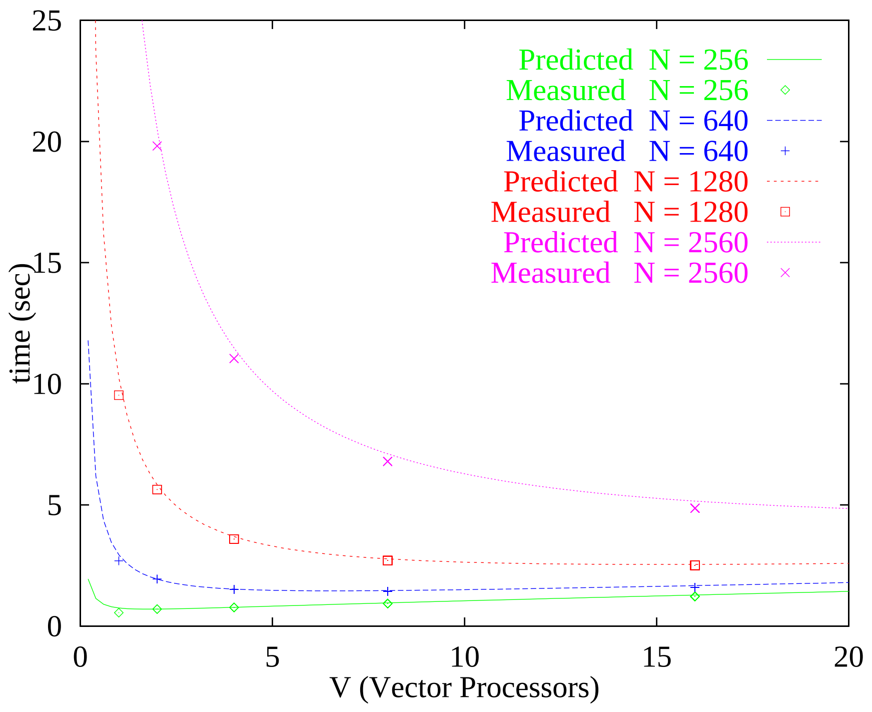
- Scalar version slowest.
- Mixed scalar and vector version fastest.

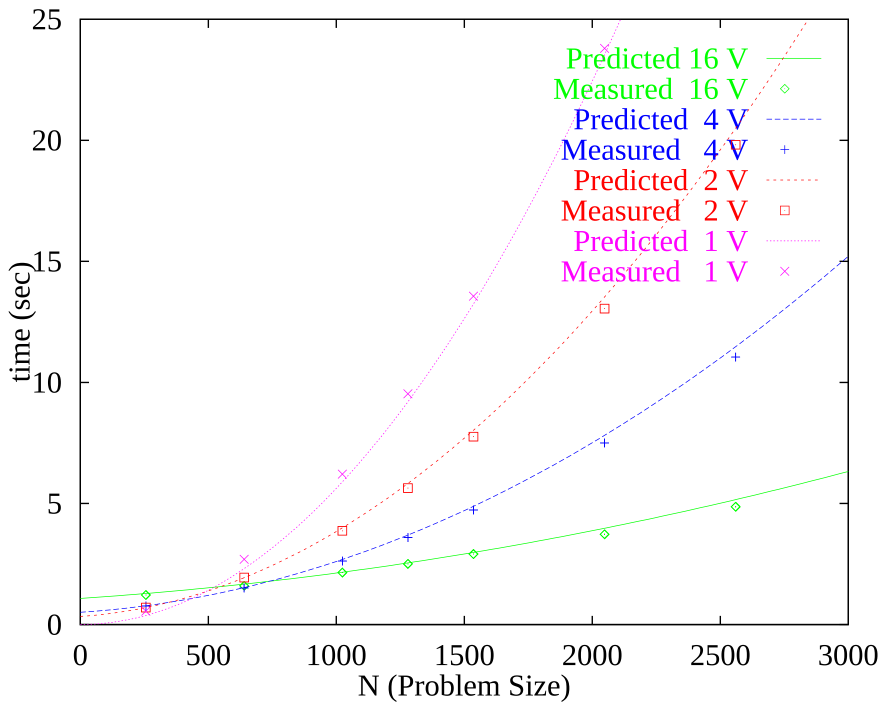
Experimental Results of Version 1



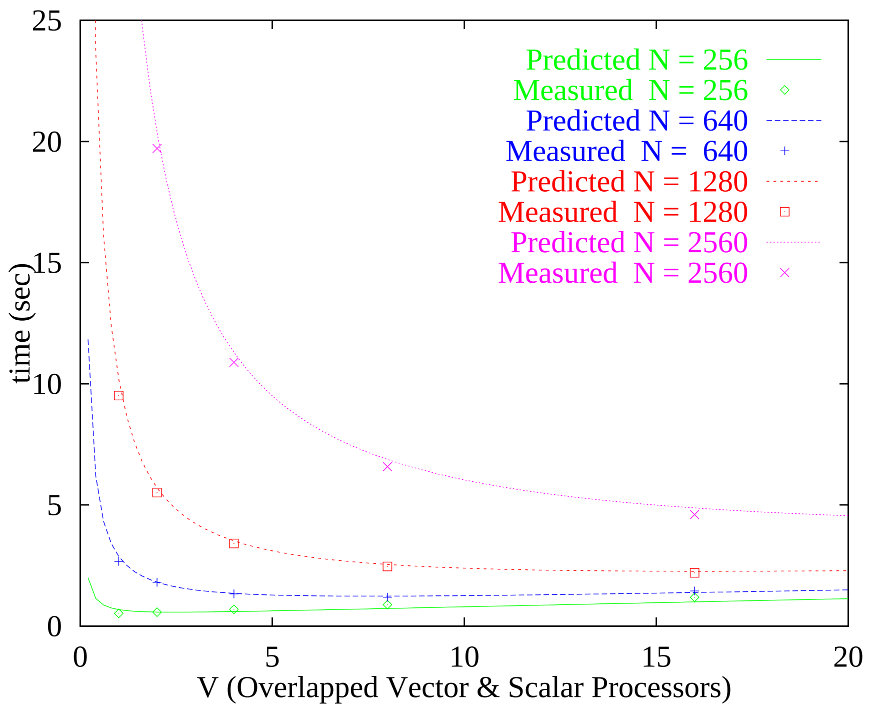


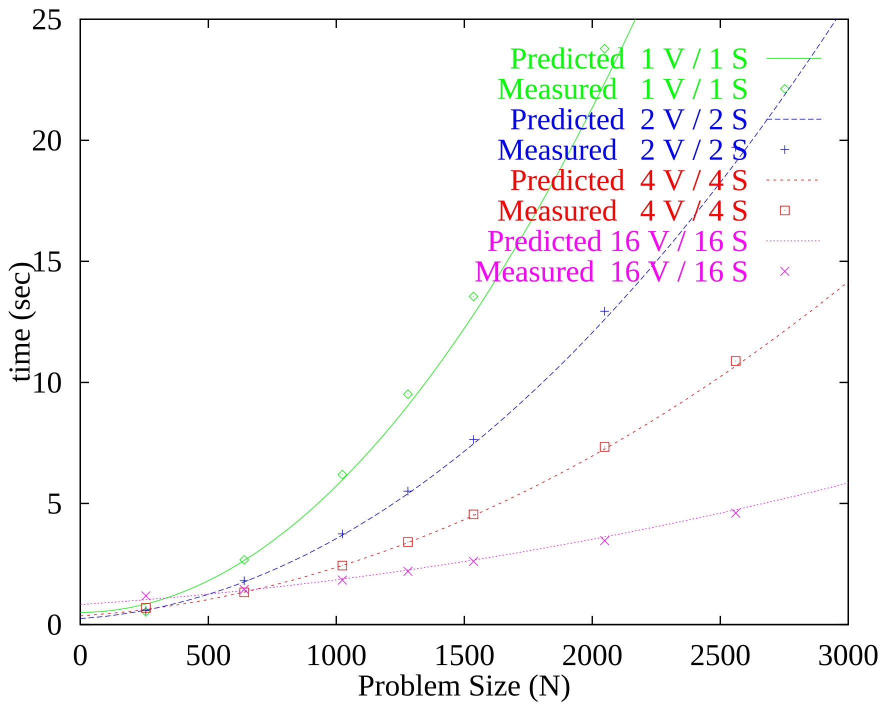
Experimental Results of Version 2



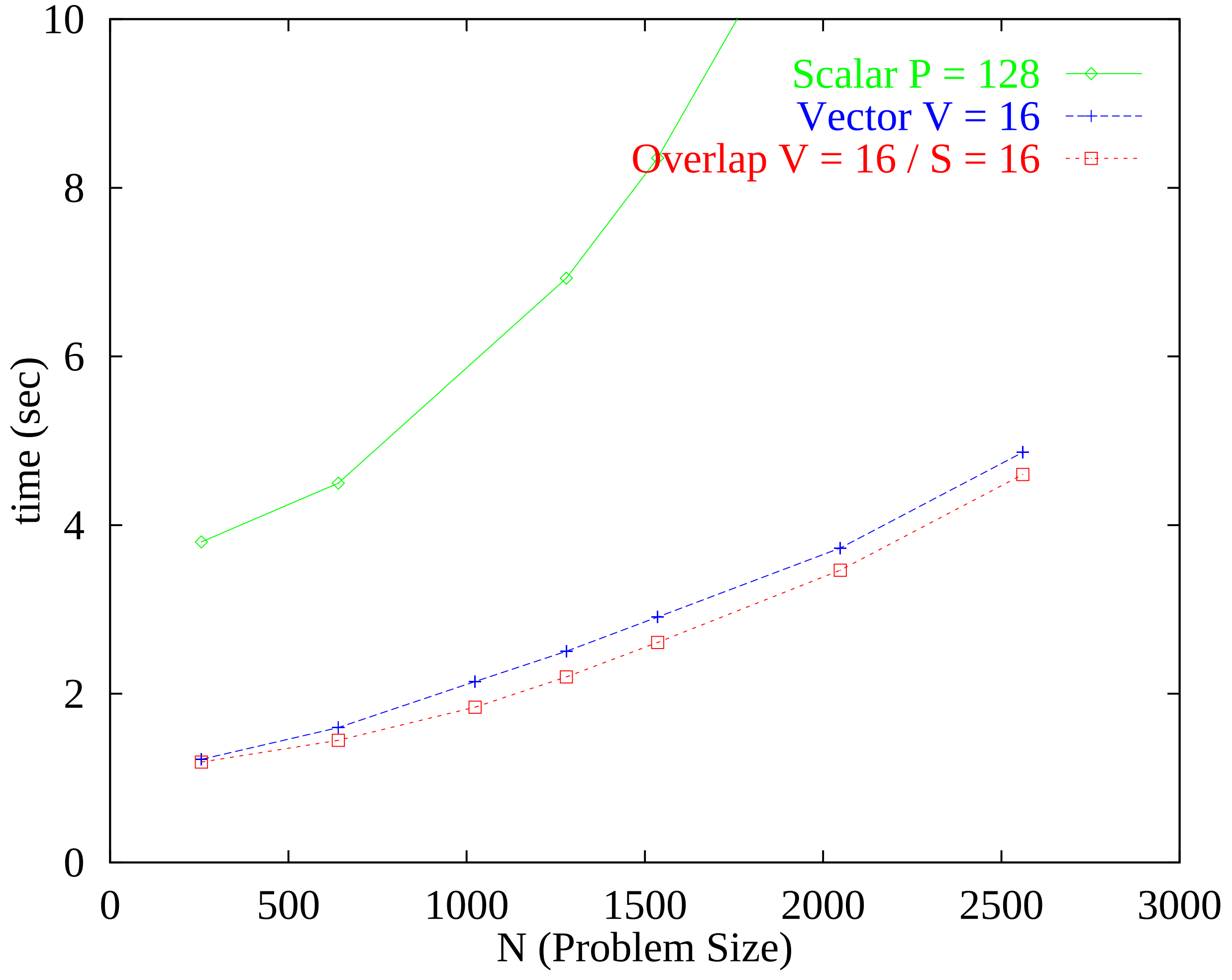


Experimental Results of Version 3





Comparative Performance of the Different Programs



Conclusions

Presented an approach for co-ordinating and organising resources in heterogeneous parallel machines.

Where:

- Different configuration structures of the machine can be easily expressed.
- Performance systematically predicted.

A pilot study demonstrated the applicability of this approach.

Acknowledgements

We would like to thank Fujitsu for providing the facilities at IFPC which made this work possible.