1

A COMPILER STRATEGY FOR SHARED VIRTUAL MEMORIES

François Bodin*, Michael O'Boyle**

**IRISA-INRIA, Campus de Beaulieu,*
35042 Rennes Cedex,
*** Department of Computer Science, University of Manchester, Oxford Rd, Manchester M13 9PL*

ABSTRACT

This paper describes a compiling strategy for SVM that exploits parallelism and minimises overheads when using an SPMD execution model. Our strategy integrates compiler techniques developed separately for data and control parallelism paradigms and integrates them into one approach. This strategy has been implemented in a prototype compiler, called MARS, currently running on the KSR-1 architecture. The technique is intended to be generic for all SVM systems. Initial results are encouraging.

1 INTRODUCTION

Shared Virtual Memory (SVM) architectures provide a single address space on distributed hardware with no physically shared memory. Instead, the distributed local memories collectively provide a virtual address space shared by all processors. SVM promises the benefit of the ease of programming found in shared-memory multiprocessors with the scalability of distributed memory multiprocessors. The compiler is relieved from the difficulties of producing optimised message-passing code, but must now be concerned with issues such as data locality and synchronisation overhead.

This paper describes a compiler strategy developed for SVM architectures and prototyped in MARS, an optimising compiler. A broad outline of the issues and costs involved is presented along with the rationale for our approach. This is followed by a brief description of successive compiler phases and samples of performance results.

1.1 Costs

The main issue in designing a parallelising compiler is the internal strategy. The major goal, for multi-processors, is to determine a set of transformations that minimise the following cost, which is an approximation to the execution time:

$$Cost = \sum_S max_P(T_{co}) + \sum_S max_P(T_{ra}) + \sum_S (T_{sy}) + \sum_S (T_{la}) + \sum_S (T_{ov})$$

where P is the set of processors and S the set of statements, T_{co} the amount of time spent performing useful computation, T_{ra} the amount of communication between processors, T_{sy} the amount of synchronisation, T_{la} the amount of communication within a processors (between cache and main memory) and T_{ov}, the amount of extra computation due to parallelisation. In general, determining the execution time is undecidable at compile time and the number of transformations is infinite. However, for a large class of interesting programs, it is possible to statically estimate each overhead and hence execution time. Furthermore, in restricting our search space by making reasonable assumptions and applying them consistently throughout the strategy, we can optimise programs in an acceptable time. Unfortunately, the costs are not independent, therefore one of the most important characteristics of a successful strategy is to trade-off competing overheads based on their relative costs which in turn depends on the target system.

1.2 SVM Characteristics

When designing a compiler strategy it is necessary to consider the main characteristics of the target platform. These characteristics determine the various costs the compiler must take into account. The main benefit of SVM is that no local/global address translation is necessary since it is provided by the OS/hardware. The page structure allows limited exploitation of locality within appropriately structured codes. However this benefit is at the price of a more complex memory model, where page movement is expensive, and characterises the issues that a compiler for SVM must consider.

There are three runtime phenomena which must be considered for efficient execution:

- **Locality of Reference:** Transformations that increase local accesses and reduce remote will reduce page movement. Once a page has been moved it is important that all it's elements are accessed before it is invalidated or replaced and will therefore determine the array access stride. Similarly, data re-use within the page should be exploited by restructuring so that re-use occurs before data is replaced.
- **False Sharing:** If two processors access a page and one of them is a writer, the page will oscillate between the two, serialising operations and greatly increasing the number of page faults. Therefore the compiler must schedule work and data in such a manner that false sharing is avoided.
- **Synchronisation:** Synchronisation is necessitated by the read/write access model of shared memory programming. This may be a considerable overhead in SVM systems [14]. An optimising compiler should place synchronisation only where necessary.

We assume a sequentially consistent memory system where optimising coherency traffic by the compiler is not directly possible and is not addressed further in this paper.

1.3 Assumptions

Certain choices are made to reduce the complexity of the compiler strategy. These assumptions are thought to give good program performance:

- **Execution Model:** The fork-and-join model allows nested parallelism but increases overhead. We use the SPMD model as it reduces overheads and enables the reduction of the synchronisation cost.
- **Data Distribution:** A knowledge of the data distribution allows remote data access optimisation. If we were to consider a model as general as HPF distribution, we may achieve an optimal solution but this will be at the expense of compiler complexity. We consider a restricted set of partition whose impact is easier to compute and simplifies later analysis such as determining cross-processor data dependence[1].

[1] A dependence whose source and sink are executed on different processors.

- **Work Scheduling:** In SVM systems the way work is scheduled to processors will determine data layout. Scheduling of work, for instance, must trade-off load balancing against communication. In our compiler we schedule work such that all write accesses are local throughout the program reducing to zero the communication overhead of writes, at the possible expense of more read accesses.

These assumptions allow successful exploitation of parallelism for programs whose behaviour is largely compile-time determinable. More dynamic programs may find these assumptions too restrictive, leading to degraded performance.

2 A COMPILER STRATEGY FOR SVM

Our compiler strategy splits naturally into two stages: global partitioning and local optimisation (called post-partitioning). The first stage makes global trade-off decisions to minimise execution time and is concerned with balancing the distinct requirements of different parts of the program. The second phase applies more aggressive optimisations once global trade-offs have been reconciled, but must trade off costs within each part of the processor program. Although distinct stages, both rely on the other making sensible choices; partitioning to minimise remote accesses depends on post-partitioning locality transformations to realise it. The post-partition stage, in turn, will try to optimise any given processor code but will only perform well if a sensible data partition is used. Hence, both stages must operate in a consistent manner.

The costs addressed by our compiler are in order of significance; parallel execution T_{co}, synchronisation T_{sy}, inter-processor communication T_{ra}, load imbalance T_{co}, intra-processor communication T_{la} and loop overhead T_{ov}. Throughout the compiler this ordering is honoured at all stages. The first four are global costs and are addressed in the global partitioning stage where an explicit trade-off is made. The last two (T_{la}, T_{ov}) are considered to be of less significance and can largely be managed once a local node program has been generated. Each compiler phase tries to reduce one or more costs. The phases of our compiler are shown in table 1 and are described in sections 3 and 4. The diagram in figure 1 shows which costs each phase attempts to reduce. Certain phases co-operate (shown by solid lines) while others require conflicting program transformations (shown by broken lines) which will trade off their associated costs.

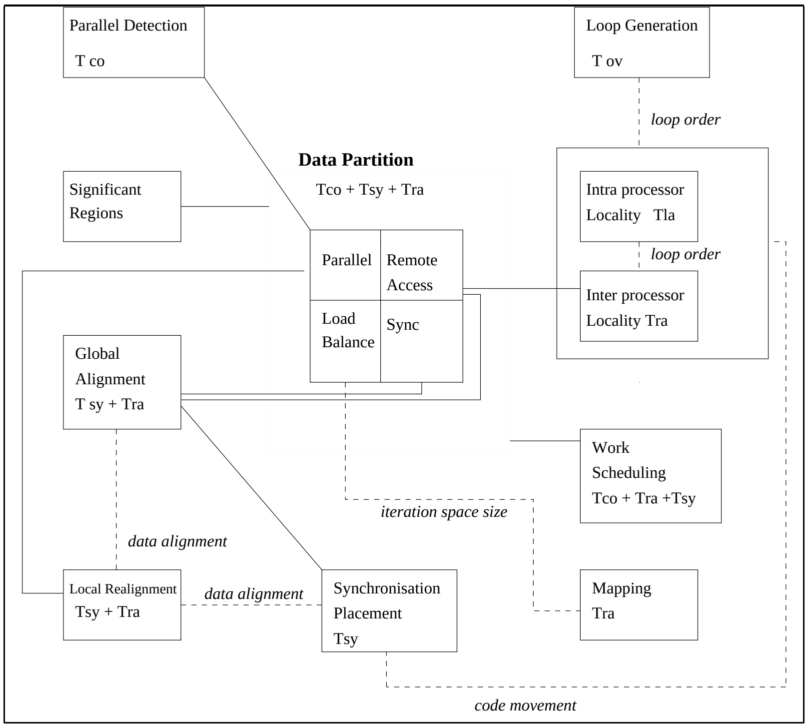

Figure 1 Relationships (solid lines) and conflicts between phases (broken lines).

3 GLOBAL PARTITIONING

At the core of the strategy is the global partitioning phase which determines the work and data distribution based on parallelism and SVM overheads. Although data cannot be physically allocated to processors at the program level, we can exploit the SVM to achieve this.

To simplify the data distribution selection, it is split into two phases: global alignment and data partitioning. To limit the search space we restrict the data distribution to orthogonal partitions of the arrays. Thus if an array has N dimensions, it may be partitioned in $2^N - 1$ different ways. Alignment has the property that corresponding array elements are always mapped to the same processor, i.e. if array element $X(6, 12)$ is mapped to processor 3, so will array elements $Y(6, 12)$, $Z(6, 12)$ etc. We can exploit this property in order to reduce

Pre-Partitioning	Parallel Detection Significant Regions
Partitioning	Global Alignment Global Data Partitioning
Post-Partitioning	Local Realignment Work Scheduling Synchronisation Placement Mapping Hierarchical Locality Loop Optimisation

Table 1 Compiler Phases

communication. If in a given loop, $Y(6,12)$, $Z(6,12)$ are both referenced, and one reference is known to be local, the other will also be local.

3.1 Global Data Alignment for SVM

Alignment transformations are applied so as to globally maximise the number of references that will be local. We use a simplified version of the method described in [13] where global alignment is restricted to permutations. In addition, we use a variant of the index space alignment algorithm [12] in which only accesses in significant regions are considered. Data alignment reduces data sharing T_{ra} and synchronisation T_{sy}, but may trade-off against parallelism T_{co}. If there is any conflict, this is addressed in the post-partitioning phase by doing local realignment of some arrays.

The alignment matrices for Loop Nests 1 are given in figure 2. For further details on their definition see [13].

Loop Nests 1

```
        DO I1 = 2 TO N1
           DO I2 = 1 TO N2
S1 :          X(I2,I1) = X(I2,I1 - 1) + Y(I1,I2)
           END DO
        END DO
        DO J1 = 1 TO N1 - 1
           DO J2 = 1 TO N2 - 1
```

```
                DO J₃ = 1 TO N₃
S₂ :                Z(J₂ + 1, J₁ + 1)   += X(J₂ + 1, J₁) + Y(J₂, J₃)
                END DO
            END DO
        END DO
```

The preferred alignment between arrays X and Y in statement $\mathcal{S}_1$, for instance, matches two dimension in statement $\mathcal{S}_1$ but only one in statement $\mathcal{S}_2$. If we assume that the global alignment phase aligns X normal to the index space and Y permuted, then we have the the Loop Nests 2; where the new alignment on Y is shown by transposing the access to array Y.

Loop Nests 2

```
        DO I₁ = 2 TO N₁
            DO I₂ = 1 TO N₂
S₁ :            X(I₂, I₁) = X(I₂, I₁ − 1) + Y(I₂, I₁)
            END DO
        END DO
        DO J₁ = 1 TO N₁ − 1
            DO J₂ = 1 TO N₂ − 1
                DO J₃ = 1 TO N₃
S₂ :                Z(J₂ + 1, J₁ + 1)   += X(J₂ + 1, J₁) + Y(J₃, J₂)
                END DO
            END DO
        END DO
```

3.2 Global Data Partitioning for SVM

Global partitioning must trade-off the conflicting requirements of different program regions. Once a decision has been reached, it is applied throughout the entire program. The approach is based on concentrating any cost analysis on the "most significant" parts of the program and determining whether a cost or transformation will have any significant impact. Given cost information provided by the following analysis, a global partitioning strategy is constructed.

Parallelism (T_{co}): Partitioning for parallelism is based on identifying array indices which can be evaluated in parallel. The parallel iterators surrounding a statement are readily available from the pre-partitioning phase. Those indices that are accessed by parallel iterators are candidates for partitioning. In Loop Nests 2, loops on $\mathtt{I}_2$, $\mathtt{J}_1$, $\mathtt{J}_2$ are parallel and loop on $\mathtt{J}_3$ is

$\mathcal{S}_1 : X(\mathtt{I}_2, \mathtt{I}_1)$	$\mathcal{S}_1 : Y(\mathtt{I}_1, \mathtt{I}_2)$	$\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
$\mathcal{S}_2 : Z(\mathtt{J}_2 + 1, \mathtt{J}_1 + 1)$	$\mathcal{S}_2 : X(\mathtt{J}_2 + 1, \mathtt{J}_1)$	$\mathcal{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
$\mathcal{S}_2 : Z(\mathtt{J}_2 + 1, \mathtt{J}_1 + 1)$	$\mathcal{S}_2 : Y(\mathtt{J}_2, \mathtt{J}_3)$	$\mathcal{A} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Figure 2 Alignment Affinity for Loop Nests 1.

partition first dimension	$\mathcal{S}_1$	Exploit Parallelism on $\mathtt{I}_2$
	$\mathcal{S}_2$	Exploit Parallelism on $\mathtt{J}_2$
partition second dimension	$\mathcal{S}_1$	Sequential execution
	$\mathcal{S}_2$	Exploit Parallelism on $\mathtt{J}_1$
partition both dimensions	$\mathcal{S}_1$	Parallelism on $\mathtt{I}_2$ limited to $\sqrt{p}$
	$\mathcal{S}_2$	Exploit Parallelism on $\mathtt{J}_1$ and $\mathtt{J}_2$

Figure 3 Impact of Partitioning on Parallelism for Loop Nests 2. Variable p is the number of processors.

a reduction. The first index of X and the two indices of Z can be computed in parallel. Figure 3 illustrates the relationship between the partitioning and parallelism.

Load Imbalance (T_{co}): If partitioning takes place with respect to "work invariant" iterators the same amount of work will take place in each processor. A parallel iterator is work invariant if it does not refer to another iterator in its bounds and is not referenced in the loop bounds of another iterator or in a conditional expression. It is possible to transform for invariance but as this may skew the data space and increase the complexity at later stages, it is not considered in this scheme. In the Loop Nests 2 example all partitions give equally good load balancing.

partition first dimension	$(\mathcal{S}_1 \delta \mathcal{S}_1)$	Local Dep	No Sync
	$(\mathcal{S}_2 \delta \mathcal{S}_2)$	Local Dep	
	$(\mathcal{S}_1 \delta \mathcal{S}_2)$	Local Dep	
partition second dimension	$(\mathcal{S}_1 \delta \mathcal{S}_1)$	Cross Dep	Sync: O(p)
	$(\mathcal{S}_2 \delta \mathcal{S}_2)$	Local Dep	
	$(\mathcal{S}_1 \delta \mathcal{S}_2)$	Cross Dep	O(1)
partition both dimensions	$(\mathcal{S}_1 \delta \mathcal{S}_1)$	Cross Dep	Sync: O($\sqrt{p}$)
	$(\mathcal{S}_2 \delta \mathcal{S}_2)$	Local Dep	
	$(\mathcal{S}_1 \delta \mathcal{S}_2)$	Cross Dep	O(1)

Figure 4 Synchronisation costs (number of synchronisations executed) for Loop Nests 2. Symbol δ denotes data dependences and symbol p the number of processors.

Synchronisation (T_{sy}): If the source and sink of a dependence are scheduled to the same processor, no synchronisation is needed. This stage uses the alignment information to determine the dimension which, if partitioned, would incur the least number of cross-processor dependences and hence synchronisations. This analysis relies on an efficient post-partition synchronisation placement phase. Figure 4 shows the impact of the partitioning decision on the number of synchronisations executed for Loop Nests 2.

Remote Accesses (T_{ra}): This analysis determines an approximate value for the remote array sections referenced for a particular partition. It determines which dimensions the arrays are to be partitioned so as to minimise non-local accesses. This partitioning scheme depends on data re-use to remove redundant non-local accesses. Figure 5 gives, for Loop Nests 2, according to the partitioning, an approximation of the amount of remote accesses assuming temporal locality is exploited.

The indices to be partitioned must be parallel and as far as costs are concerned, it is assumed that synchronisation is more costly than remote accesses; this is based on experience with KSR-1 but may change with other systems. Minimising remote accesses is more important than load imbalance; this is based on the assumption that a non-local access costs more than one computation assuming no latency tolerance. This is simplistic and the dominant costs may vary. With current strategy, for Loop Nests 2, the partition of the first dimension on the arrays would be chosen.

partition first dimension	$\mathcal{S}_1 : X(\mathtt{I}_2, \mathtt{I}_1 - \mathtt{1})$	
	$\mathcal{S}_1 : Y(\mathtt{I}_2, \mathtt{I}_1)$	
	$\mathcal{S}_2 : X(\mathtt{J}_2 + 1, \mathtt{J}_1)$	
	$\mathcal{S}_2 : Y(\mathtt{J}_3, \mathtt{J}_2)$	$\frac{\mathtt{N}_3\mathtt{N}_2}{p}$
partition second dimension	$\mathcal{S}_1 : X(\mathtt{I}_2, \mathtt{I}_1 - \mathtt{1})$	$\mathtt{N}_1$
	$\mathcal{S}_1 : Y(\mathtt{I}_2, \mathtt{I}_1)$	
	$\mathcal{S}_2 : X(\mathtt{J}_2 + 1, \mathtt{J}_1)$	$\mathtt{N}_1$
	$\mathcal{S}_2 : Y(\mathtt{J}_3, \mathtt{J}_2)$	$\mathtt{N}_3\mathtt{N}_2$
partition both dimensions	$\mathcal{S}_1 : X(\mathtt{I}_2, \mathtt{I}_1 - \mathtt{1})$	$\frac{\mathtt{N}_1}{\sqrt{p}}$
	$\mathcal{S}_1 : Y(\mathtt{I}_2, \mathtt{I}_1)$	
	$\mathcal{S}_2 : X(\mathtt{J}_2 + 1, \mathtt{J}_1)$	$\frac{\mathtt{N}_1}{\sqrt{p}}$
	$\mathcal{S}_2 : Y(\mathtt{J}_3, \mathtt{J}_2)$	$\frac{\mathtt{N}_3\mathtt{N}_2}{\sqrt{p}}$

Figure 5 Remote access costs for Loop Nests 2. Variable p is the number of processors.

4 POST-PARTITION STAGE

Determining a good data partition will not in itself provide a successful implementation. The post-partition phase is responsible for reducing synchronisation, communication and loop overheads according to previous decisions. This phase is concerned with minimising the costs of implementing parallelism:

Local Realignment: Data partitioning for significant regions may be at the expense of parallelism in other regions for lower dimensional arrays. If a lower dimensional array is aligned on non-partitioned indices of the template, then all accesses to it will be sequentialised. To limit the loss of parallelism, we realign lower dimensional arrays onto partitioned template indices if applicable. This allows exploitation of parallelism in non-significant regions at the possible expense of remote accesses and synchronisations. This tries to reduce T_{co} in certain instances, but trades off against global alignment and may increase T_{ra} and T_{sy}.

Work Scheduling: The scheduling of work to processors given a data distribution depends on machine costs and the statement structure. At present we only consider creation and reduction parallelism. This stage marks each statement with the form of parallelism (if any) to be exploited in each partitioned index. For many statements there is only one form of parallelism present, i.e. summing an array into a scalar only has reduction parallelism, while initialising an array may only exhibit creation parallelism. In those cases where both are present, by default we schedule for creation parallelism as reductions always incur at least one barrier. Future implementations will be more sophisticated in determining the appropriate schedule.

Synchronisation: Synchronisation points can be removed by detecting those dependences that are entirely local. Once we have computed the minimum number of cross-processor dependences (using alignment information), they are mapped on the control flow graph. We then try to find the best place to insert synchronisations so as to minimise the number executed at runtime [14]. The efficacy of this phase depends on the data distribution determined by previous alignment, partitioning and local realignment phases.

Mapping: Once the data partition and synchronisation points have been determined we generate the local iteration space and new program structure depending on the form of parallelism determined previously. Although the local iteration space is based on the data layout, in SVM systems it is the actual computation executed at runtime that determines the layout of data; data distribution is "virtual" since actual data allocation is automatically handled by the SVM. This stage can reduce false sharing ,T_{ra}, by ensuring that data and work is partitioned on page boundaries. This may increase the work ,T_{co}, performed in some processors.

Hierarchical Locality Optimisation: Communication cost and memory hierarchy usage is optimised at this stage [7, 15, 16]. Data re-use and spatial locality require a combination of strip-mining, skewing, interchange and the introduction of temporaries. Minimising cache to local memory traffic requires the same treatment as reducing remote accesses. As remote accesses are an order of magnitude more expensive, we minimise them first at the possible expense of more cache misses. The loop overhead optimisations interact with the locality optimisations since they may both require conflicting loop orders. However, in SVM systems we believe page faults to be more important than loop overhead and thus the loop optimisation transformations assume a fixed loop order.

Loop Optimisation: This step's goal is to reduce the loop management overhead that may have been introduced by previous steps. Partitioning work increases the number of loop bounds and conditionals to evaluate before executing the loop body. In certain programs this can be a significant overhead particularly if strip-mining has been used to exploit locality. This is the final optimisation as it is considered less important and is concerned with uni-processor performance rather than parallel performance. This stage only improves T_{ov} and may trade off against the hierarchical locality optimisation stage as they may have conflicting loop order requirements.

5 RESULTS

The MARS compiler implements the global strategy described above. It is built on top of Sage++ [4] and uses an augmented linear algebraic framework. The framework is designed to simplify and unify the implementation of the compiler phases. We applied our compiler strategy to two Fortran programs, Tred2 and the ADI fragment used in [8]. The resulting code was run on a KSR-1 and compared with the results of the auto-paralleliser KAP. Tred2 is a 148 lines Eispack routine, while the ADI fragment is 19 lines in length. Tred2 allows the compilers to be tested over a relatively complex program with many trade-offs. ADI on the other hand is much simpler, but requires a consistent parallelisation strategy. Our compiler outperforms KAP, by an order of magnitude for large data sizes for Tred2. This effect is largely due to excessive unnecessary synchronisations within loop bodies. The performance gap is less for the ADI program, as it is relatively trivial to parallelise. The differences are due to data movement, due to the opposing partitioning of two loop nests. The MARS compiler automatically determines a partition which requires no data movement or synchronisation.

6 RELATED WORK

Anderson et al. [1] proposed a global approach to optimise parallelism and locality. The approach is based on finding a data partition and the corresponding iteration space distribution over the processors. In this work the only costs considered are communication and parallelism; synchronisation and access stride are ignored. Furthermore exploiting locality is restricted to finding communication free partitions (which may not exist) rather than those that minimise

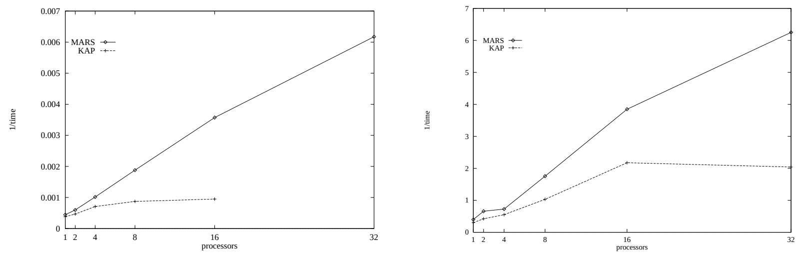

Figure 6 Performance for Tred2, n=512 (left). Performance for ADI, n=512 (right). The x-axis is the number of processors, the y-axis gives is 1/time.

remote accesses. Also, the relationship between data partitioning and post-partitioning transformations is not explored.

Appelbe et al. [2] approach relies on partitioning loop iteration space on processors. With the set of parallel loop a set of "distributed array references" is computed. Using a set of locality constraints, expressed in a graph which nodes are loops and arcs locality constraints the partitioning of the iteration spaces is decided. The code generation algorithm also consider array padding for eliminating false-sharing. Insertion of synchronisations happened only when locality constraints are not satisfied (i.e. cross-processor data dependences) and it is based on Kennedy/Callahan algorithm [18]. Our work differs in many aspects. First we exploit the properties of the owner computes rule for mapping computations and we consider both locality and synchronisation costs.

Gupta et al. [9] have proposed a system based on the Component Affinity Graph [12]. The partitioning decision is taken according to an approximation of the amount of communication. This approach has also the problem of not integrating other costs, such as synchronisation, when taking decision and considering the impact on post-partition transformations.

7 CONCLUSION AND FUTURE WORK

This paper describes an optimising compiler strategy for SVM systems. At present the compiler uses a static data partitioning approach to parallelism exploitation. However, internal compiler structure allows data to be redistributed at every statement and we are currently developing analysis to determine when redistribution (possibly by prefetching and post-storing) is worthwhile. There are occasions when loop based scheduling is preferable to a data based approach, particularly with regard to load balancing; we hope to address this issue in the future. Presently our strategy does not include interprocedural analysis, but this will be added shortly. We will also investigate how compile time knowledge can be used to relax memory coherence and how this may be integrated into a consistent compiler strategy.

REFERENCES

[1] Anderson J.M. and Lam M.S. **Global Optimizations for Parallelism and Locality on Scalable Parallel Machines**, Proceedings of Programming Languages Design and Implementation, ACM Press, 1993.

[2] Appelbe B. Doddapaneni S. and Hardnett C. **A new Algorithm for Global Optimization for Parallelism and Locality**, LNCS 892, 7th International Workshop on Languages and Compilers for Parallel Computing, Ithaca, NY, August, 1994.

[3] Beckner S. Chapman B. and Zima H. **Vienna Fortran 90**, Proceedings of Scalable High Performance Computing Conference, 1992.

[4] Bodin F. Beckman P. Gannon D. and Srinivas J. **Sage++: A Class Library for Building Fortran and C++ Restructuring Tools**, Second Object-Oriented Numerics Conference, Oregon (USA), April 1994.

[5] Bodin F. Granston E. and Montaut T. **Evaluating Two Loop Transformations for Reducing Multiple-Writer False Sharing**, LCPC, Springer-Verlag in the Lecture Notes in Computer Science, August 1994.

[6] Bodin F. Eisenbeis C. Jalby W. and Windheiser D. **A Strategy for Array Management in Local Memory**, Special Issue of Math. Programming B on Applications of Discrete Programming in Computer Science, 1992.

[7] Callahan D. Carr S. and Kennedy K. **Improving Register Allocation for Subscripted Variables**, Proceedings of the Conference on Programming Language Design and Implementation, 1990.

[8] Choudhary A. Fox G. Hiranandani S. Kennedy K. Koelbel C. Ranka S. and Tseng C. **Unified Compilation of Fortran 77D and 90D**, ACM Letters on Programming Languages and Systems Vol 2, N 1-4, March-December, 1993.

[9] Gupta M. and Banerjee P. **Paradigm: A Compiler for Automatic Data Distribution on Multicomputers**, Proceedings of the International Conference on Supercomputing, pp 87-96, 1993.

[10] Lahjomri Z. and Priol T. **Koan : A Shared Virtual Memory for the iPSC/2 Hypercube**, Proceedings of CONPAR/VAPP92, Lyon, September, 1992.

[11] Lam M. Rothberg E. and Wolf M.E. **The Cache Performance and Optimizations of Blocked Algorithms**, Proceedings of the Fourth ACM ASPLOS conference, April 91.

[12] Li J. and Chen M. **Index Domain Alignment: Minimizing Cost of Cross-Referencing between Distributed Arrays**, IEEE Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation, October 1990

[13] O'Boyle M.F.P. and Hedayat G.A. **Data Alignment: Transformations to Reduce Communication on Distributed Memory Architectures**, Scalable High Performance Computing Conference, Williamsburg, April 1992.

[14] O'Boyle M.F.P. Kervella L. and Bodin F. **Synchronisation Minimisation in a SPMD execution model**, *to appear* in Journal of Parallel and Distributed Computing special issue on Distributed Shared Memory Systems.

[15] Wolf M.E. and Lam M. **A Data Locality Optimizing Algorithm**, ACM Conference on Programming Language Design and Implementation, June 26-28, 1991.

[16] Wolf M.E. and Lam M. **A Loop Transformation Theory and an Algorithm to Maximize Parallelism**, IEEE Transactions on Parallel and Distributed Systems, Vol 2, No 4, October 1991.

[17] Wolfe M.J. **Optimizing Supercompilers for Supercomputers**, PhD thesis, University of Illinois, October 1982.

[18] Zima H. **Supercompilers for Parallel and Vector Computers**, ACM Press, 1991.