Expressing Irregular Computations in Modern Fortran Dialects*

Jan F. Prins, Siddhartha Chatterjee, and Martin Simons

Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599-3175
{prins,sc,simons}@cs.unc.edu

Abstract. Modern dialects of Fortran enjoy wide use and good support on high-performance computers as performance-oriented programming languages. By providing the ability to express nested data parallelism in Fortran, we enable irregular computations to be incorporated into existing applications with minimal rewriting and without sacrificing performance within the regular portions of the application. Since performance of nested data-parallel computation is unpredictable and often poor using current compilers, we investigate source-to-source transformation techniques that yield Fortran 90 programs with improved performance and performance stability.

1 Introduction

Modern science and engineering disciplines make extensive use of computer simulations. As these simulations increase in size and detail, the computational costs of naive algorithms can overwhelm even the largest parallel computers available today. Fortunately, computational costs can be reduced using sophisticated modeling methods that vary model resolution as needed, coupled with sparse and adaptive solution techniques that vary computational effort in time and space as needed. Such techniques have been developed and are routinely employed in sequential computation, for example, in cosmological simulations (using adaptive n-body methods) and computational fluid dynamics (using adaptive meshing and sparse linear system solvers).

However, these so-called irregular or unstructured computations are problematic for parallel computation, where high performance requires equal distribution of work over processors and locality of reference within each processor. For many irregular computations, the distribution of work and data cannot be characterized *a priori*, as these quantities are input-dependent and/or evolve with the computation itself. Further, irregular computations are difficult to express using performance-oriented languages such as Fortran, because there is an apparent mismatch between data types such as trees, graphs, and nested sequences characteristic of irregular computations and the statically analyzable rectangular multi-dimensional arrays that are the core data types in modern

* This research was supported in part by NSF Grants #CCR-9711438 and #INT-9726317. Chatterjee is supported in part by NSF CAREER Award #CCR-9501979. Simons is on leave from TU Berlin and is supported by a scholarship from the German Research Foundation (DFG).

Fortran dialects such as Fortran 90/95 [19], and High Performance Fortran (HPF) [16]. Irregular data types can be introduced using the data abstraction facilities, with a representation exploiting pointers. Optimization of operations on such an abstract data type is currently beyond compile-time analysis, and compilers have difficulty generating high-performance parallel code for such programs. This paper primarily addresses the expression of irregular computations in Fortran 95, but does so with a particular view of the compilation and high performance execution of such computations on parallel processors.

The modern Fortran dialects enjoy increasing use and good support as mainstream performance-oriented programming languages. By providing the ability to express irregular computations as Fortran modules, and by preprocessing these modules into a form that current Fortran compilers can successfully optimize, we enable irregular computations to be incorporated into existing applications with minimal rewriting and without sacrificing performance within the regular portions of the application.

For example, consider the NAS CG (Conjugate Gradient) benchmark, which solves an unstructured sparse linear system using the method of conjugate gradients [2]. Within the distributed sample sequential Fortran solution, 79% of the lines of code are standard Fortran 77 concerned with problem construction and performance reporting. The next 16% consist of scalar and regular vector computations of the BLAS 2 variety [17], while the final 5% of the code is the irregular computation of the sparse matrix-vector product. Clearly we want to rewrite only this 5% of the code (which performs 97% of the work in the class B computation), while the remainder should be left intact for the Fortran compiler. This is not just for convenience. It is also critical for performance reasons; following Amdahl's Law, as the performance of the irregular computation improves, the performance of the regular component becomes increasingly critical for sustained high performance overall. Fortran compilers provide good compiler/annotation techniques to achieve high performance for the regular computations in the problem, and can thus provide an efficient and seamless interface between the regular and irregular portions of the computation.

We manually applied the implementation techniques described in Sect. 4 to the irregular computation in the NAS CG problem. The resultant Fortran program achieved a performance on the class B NAS CG 1.0 benchmark of 13.5 GFLOPS using a 32 processor NEC SX-4 [25]. We believe this to be the highest performance achieved for this benchmark to date. It exceeds, by a factor of 2.6, the highest performance reported in the last NPB 1.0 report [27], and is slightly faster than the 12.9 GFLOPS recently achieved using a 1024 processor Cray T3E-900 [18]. These encouraging initial results support the thesis that high-level expression and high-performance for irregular computations can be supported simultaneously in a production Fortran programming environment.

2 Expressing irregular computations using nested data parallelism

We adopt the data-parallel programming model of Fortran as our starting point. The data-parallel programming model has proven to be popular because of its power and simplicity. Data-parallel languages are founded on the concept of collections (such as arrays) and a means to allow programmers to express parallelism through the applica-

tion of an operation independently to all elements of a collection (e.g., the elementwise addition of two arrays). Most of the common data-parallel languages, such as the array-based parallelism of Fortran 90, offer restricted data-parallel capabilities: they limit collections to multidimensional rectangular arrays, limit the type of the elements of a collection to scalar and record types, and limit the operations that can be applied in parallel to the elements of a collection to certain predefined operations rather than arbitrary user-defined functions. These limitations are aimed at enabling compile-time analysis and optimization of the work and communication for parallel execution, but make it difficult to express irregular computations in this model.

If the elements of a collection are themselves permitted to have arbitrary type, then arbitrary functions can be applied in parallel over collections. In particular, by operating on a collection of collections, it is possible to specify a parallel computation, each simultaneous operation of which in turn involves (a potentially different-sized) parallel subcomputation. This programming model, called *nested data parallelism*, combines aspects of both data parallelism and control parallelism. It retains the simple programming model and portability of the data-parallel model while being better suited for describing algorithms on irregular data structures. The utility of nested data parallelism as an expressive mechanism has been understood for a long time in the LISP, SETL [29], and APL communities, although always with a sequential execution semantics and implementation.

Nested data parallelism occurs naturally in the succinct expression of many irregular scientific problems. Consider the sparse matrix-vector product at the heart of the NAS CG benchmark. In the popular compressed sparse row (CSR) format of representing sparse matrices, the nonzero elements of an $m \times n$ sparse matrix `A` are represented as a sequence of m rows $[R_1, \ldots, R_m]$, where the ith row is, in turn, represented by a (possibly empty) sequence of (v, c) pairs where v is the nonzero value and $1 \leq c \leq n$ is the column in which it occurs: $R_i = [(v_1^i, c_1^i), \ldots, (v_{k_i}^i, c_{k_i}^i)]$. With a dense n-vector x represented as a simple sequence of n values, the sparse matrix-vector product $y = Ax$ may now be written as shown using the NESL notation [4]:

$$y = \{\texttt{sparse\_dot\_product}(R,x)\ :\ R\ \texttt{in}\ A\}.$$

This expression specifies the application of `sparse_dot_product`, in parallel, to each row of A to yield the m element result sequence y. The sequence constructor `{...}` serves a dual role: it specifies parallelism (for each R in A), and it establishes the order in which the result elements are assembled into the result sequence, i.e., $y_i = \texttt{sparse\_dot\_product}(R_i, x)$. We obtain nested data parallelism if the body expression $\texttt{sparse\_dot\_product}(R, x)$ itself specifies the parallel computation of the dot product of row R with x as the sum-reduction of a sequence of nonzero products:

$$\texttt{function sparse\_dot\_product}(R,x) = \texttt{sum}(\{v\texttt{*}x[c]\texttt{: }(v,c)\ \texttt{in}\ R\})$$

More concisely, the complete expression could also written as follows:

$$y = \{\texttt{sum}(\{v\texttt{*}x[c]\texttt{: }(v,c)\ \texttt{in}\ R\})\texttt{: }R\ \texttt{in}\ A\}$$

where the nested parallelism is visible as nested sequence constructors in the source text.

```
MODULE Sparse_matrices

  IMPLICIT none

  TYPE Sparse_element
     REAL                                          :: val
     INTEGER                                       :: col
  END TYPE Sparse_element

  TYPE Sparse_row_p
     TYPE (Sparse_element), DIMENSION (:), POINTER :: elts
  END TYPE Sparse_row_p

  TYPE Sparse_matrix
     INTEGER                                       :: nrow, ncol
     TYPE (Sparse_row_p), DIMENSION (:), POINTER   :: rows
  END TYPE Sparse_matrix

END MODULE Sparse_matrices
```

Fig. 1 Fortran 90 definition of a nested sequence type for sparse matrices

Nested data parallelism provides a succinct and powerful notation for specifying parallel computation, including irregular parallel computations. Many more examples of efficient parallel algorithms expressed using nested data parallelism have been described in [4].

3 Nested data parallelism in Fortran

If we consider expressing nested data parallelism in standard imperative programming languages, we find that they either lack a data-parallel control construct (C, C++) or else lack a nested collection data type (Fortran). A data-parallel control construct can be added to C [11] or C++ [30], but the pervasive pointer semantics of these languages complicate its meaning. There is also incomplete agreement about the form of parallelism should take in these languages.

The `FORALL` construct, originated in HPF [16] and later added into Fortran 95, specifies data-parallel evaluation of expressions and array assignments. To ensure that there are no side effects between these parallel evaluations, functions that occur in the expressions must have the `PURE` attribute. Fortran 90 lacks a construct that specifies parallel evaluations. However, many compilers infer such an evaluation if specified using a conventional `DO` loop, possibly with an attached directive asserting the independence of iterations. `FORALL` constructs (or Fortran 90 loops) may be nested. To specify nested data-parallel computations with these constructs, it suffices to introduce nested aggregates, which we can do via the data abstraction mechanism of Fortran 90.

As a consequence of these language features, it is entirely possible to express nested data-parallel computations in modern Fortran dialects. For example, we might introduce

```
SUBROUTINE smvp(a, x, y)

  USE Sparse_matrices, ONLY : Sparse_matrix

  IMPLICIT none

  TYPE (Sparse_matrix), INTENT(IN) :: a
  REAL, DIMENSION(:), INTENT(IN)   :: x
  REAL, DIMENSION(:), INTENT(OUT)  :: y

  FORALL (i = 1:a%nrow)

     y(i) = SUM(a%rows(i)%elts%val * x(a%rows(i)%elts%col))

  END FORALL

END SUBROUTINE smvp
```

Fig. 2 Use of the derived type `Sparse_matrix` in sparse matrix-vector product.

the types shown in Fig. 1 to represent a sparse matrix. `Sparse_element` is the type of a sparse matrix element, i.e., the (v, c) pair of the NESL example. `Sparse_row_p` is the type of vectors (1-D arrays) of sparse matrix elements, i.e., a row of the matrix. A sparse matrix is characterized by the number of rows and columns, and by the nested sequence of sparse matrix elements.

Using these definitions, the sparse matrix-vector product can be succinctly written as shown in Fig. 2. The `DO` loop specifies parallel evaluation of the inner products for all rows. Nested parallelism is a consequence of the use of parallel operations such as sum and elementwise multiplication, projection, and indexing.

Discussion

Earlier experiments with nested data parallelism in imperative languages include V [11], Amelia [30], and F90V [1]. For the first two of these languages the issues of side-effects in the underlying notation (C++ and C, respectively) were problematic in the potential introduction of interference between parallel iterations, and the efforts were abandoned. Fortran finesses this problem by requiring procedures used within a `FORALL` construct to be `PURE`, an attribute that can be verified statically. This renders invalid those constructions in which side effects (other than the nondeterministic orders of stores) can be observed, although such a syntactic constraint is not enforced in Fortran 90.

The specification of nested data parallelism in Fortran and NESL differ in important ways, many of them reflecting differences between the imperative and functional programming paradigms.

First, a sequence is formally a function from an index set to a value set. The NESL sequence constructor specifies parallelism over the value set of a sequence while the Fortran `FORALL` statement specifies parallelism over the index set of a sequence. This

allows a more concise syntax and also makes explicit the shape of the common index domain shared by several collections participating in a `FORALL` construct.

Second, the NESL sequence constructor implicitly specifies the ordering of result elements, while this ordering is explicit in the `FORALL` statement. One consequence is that the restriction clause has different semantics. For instance, the NESL expression

$$v = \{i\texttt{: } i \texttt{ in } [1\texttt{:}n] \texttt{ | oddp(}i\texttt{) }\}$$

yields a result sequence v of length $\lfloor n/2 \rfloor$ of odd values while the Fortran statement

$$\texttt{FORALL (}i \texttt{ = } 1\texttt{:}n\texttt{, odd(}i\texttt{)) } v\texttt{(}i\texttt{) = } i$$

replaces the elements in the odd-numbered positions of v.

Third, the Fortran `FORALL` construct provides explicit control over memory. Explicit control over memory can be quite important for performance. For example, if we were to repeatedly multiply the same sparse matrix repeatedly by different right hand sides (which is in fact exactly what happens in the CG benchmark), we could reuse a single temporary instead of freeing and allocating each time. Explicit control over memory also gives us a better interface to the regular portions of the computation.

Finally, the base types of a nested aggregate in Fortran are drawn from the Fortran data types and include multidimensional arrays and pointers. In NESL, we are restricted to simple scalar values and record types. Thus, expressing a sparse matrix as a collection of supernodes would be cumbersome in NESL. Another important difference is that we may construct nested aggregates of heterogeneous depth with Fortran, which is important, for example, in the representation of adaptive oct-tree spatial decompositions.

4 Implementation issues

Expression of nested data-parallelism in Fortran is of limited interest and of no utility if such computations can not achieve high performance. Parallel execution and tuning for the memory hierarchy are the two basic requirements for high performance. Since the locus of activity and amount of work in a nested data-parallel computation can not be statically predicted, run-time techniques are generally required.

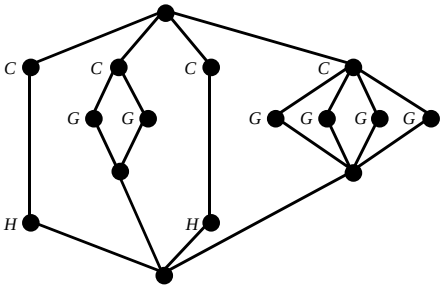
4.1 Implementation strategies

There are two general strategies for the parallel execution of nested data parallelism, both consisting of a compile-time and a run-time component.

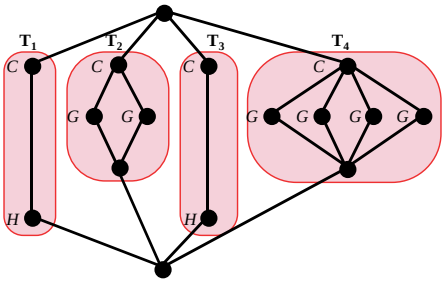
The thread-based approach. This technique conceptually spawns a different thread of computation for every parallel evaluation within a `FORALL` construct. The compile-time component constructs the threads from the nested loops. A run-time component dynamically schedules these threads across processors. Recent work has resulted in run-time scheduling techniques that minimize completion time and memory use of the generated threads [9, 6, 20]. Scheduling very fine-grained threads (e.g., a single multiplication in the sparse matrix-vector product example) is impractical, hence compile-time techniques are required to increase thread granularity, although this may result in lost parallelism and increased load imbalance.

```
FORALL (i = 1:4)
      WHERE C(i) DO
            FORALL (j = 1:i) DO
                  G(i,j)
            END FORALL
      ELSEWHERE
            H(i)
      END WHERE
END FORALL
```

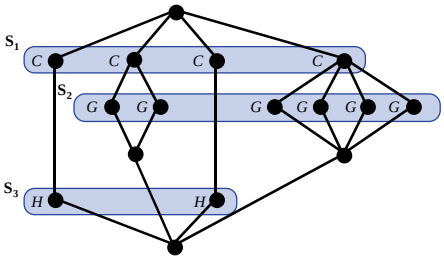
(a)



(b)



(c)



(d)

Fig. 3 (a) Nested data-parallel program. (b) The associated dependence graph. (c) Thread decomposition of the graph. (d) Data-parallel decomposition of the graph.

The flattening approach. This technique replaces nested loops by a sequence of steps, each of which is a simple data-parallel operation. The compile-time component of this approach is a program transformation that replaces `FORALL` constructs with "data-parallel extensions" of their bodies and restructures the representation of nested aggregate values into a form suitable for the efficient implementation of the data-parallel operations [8, 26]. The run-time component is a library of data-parallel operations closely resembling HPFLIB, the standard library that accompanies HPF. A nested data-parallel loop that has been flattened may perform a small multiplicative factor of additional work compared with a sequential implementation. However, full parallelism and optimal load balance are easily achieved in this approach. Compile-time techniques to fuse data-parallel operations can reduce the number of barrier synchronizations, decrease space requirements, and improve reuse [12, 24].

The two approaches are illustrated for a nested data-parallel computation and its associated dependence graph[1] in Fig. 3. Here `G` and `H` denote assignment statements that can not introduce additional dependences, since there can be no data dependences between iterations of `FORALL` loops.

In Fig. 3(c) we show a decomposition of the work into parallel threads $T_1, \ldots, T_4$. In this decomposition the body of the outer `FORALL` loop has been serialized to increase the grain size of each thread. As a result the amount of work in each thread is quite

[1] We are using HPF `INDEPENDENT` semantics for the control dependences of a `FORALL` loop.

different. On the other hand, since each thread executes a larger portion of the sequential implementation, it can exhibit good locality of reference.

In Fig. 3(d) we show a decomposition of the work into sequential steps $S_1, \ldots, S_3$, each of which is a simple data-parallel operation. The advantage of this approach is that we may partition the parallelism in each operation to suit the resources. For example, we can create parallel slack at each processor to hide network or memory latencies. In this example, the dependence structure permits the parallel execution of steps S_1 and S_2, although this increases the complexity of the run time scheduler.

4.2 Nested parallelism using current Fortran compilers

What happens when we compile the Fortran 90 sparse matrix-vector product `smvp` shown in Fig. 2 for parallel execution using current Fortran compilers?

For shared-memory multiprocessors we examined two auto-parallelizing Fortran 90 compilers: the SGI F90 V7.2.1 compiler (beta release, March 1998) for SGI Origin class machines and the NEC FORTRAN90/SX R7.2 compiler (release 140, February 1998) for the NEC SX-4. We replaced `FORALL` construct in Fig. 2 with an equivalent `DO` loop to obtain a Fortran 90 program. Since the nested parallel loops in `smvp` do not define a polyhedral iteration space, many classical techniques for parallelization do not apply. However, both compilers recognize that iterations of the outer loop (over rows) are independent and, in both cases, these iterations are distributed over processors. The dot-product inner loop is compiled for serial execution or vectorized. This strategy is not always optimal, since the distribution of work over outermost iterations may be uneven or there may be insufficient parallelism in the outer iterations.

For distributed memory multiprocessors we examined one HPF compiler. This compiler failed to compile `smvp` because it had no support for pointers in Fortran 90 derived types. Our impression is that this situation is representative of HPF compilers in general, since the focus has been on the parallel execution of programs operating on rectangular arrays. The data distribution issues for the more complex derived types with pointers are unclear. Instead, HPF 2.0 supports the non-uniform distribution of arrays over processors. This requires the programmer to embed irregular data structures in an array and determine the appropriate mapping for the distribution.

We conclude that current Fortran compilers do not sufficiently address the problems of irregular nested data parallelism. The challenge for irregular computations is to achieve uniformly high and predictable performance in the face of dynamically varying distribution of work. We are investigating the combined use of threading and flattening techniques for this problem.

Our approach is to transform nested data parallel constructs into simple Fortran 90, providing simple integration with regular computations, and leveraging the capabilities of current Fortran compilers. This source-to-source translation restricts our options somewhat for the thread scheduling strategy. Since threads are not part of Fortran 90, the only mechanism for their (implicit) creation are loops, and the scheduling strategies we can choose from are limited by those offered by the compiler/run-time system. In this regard, standardized loop scheduling directives like the OpenMP directives [23] can improve portability.

A nested data parallel computation should be transformed into a (possibly nested) iteration space that is partitioned over threads. Dynamic scheduling can be used to tolerate variations in progress among threads. Flattening of the loop body can be used to ensure that the amount of work per thread is relatively uniform.

4.3 Example

Consider a sparse $m \times n$ matrix A with a total of r nonzeros. Implementation of the simple nested data parallelism in the procedure `smvp` of Fig. 2 must address many of the problems that may arise in irregular computations:

- Uneven units of work: A may contain both dense and sparse rows.
- Small units of work: A may contain rows with very few nonzeros.
- Insufficient units of work: if n is less than the number of processors and r is sufficiently large, then parallelism should be exploited within the dot products rather than between the dot products.

We constructed two implementations of `smvp`. The *pointer-based* implementation is obtained by direct compilation of the program in Fig. 2 using auto-parallelization. As mentioned, this results in a parallelized outer loop, in which the dot products for different rows are statically or dynamically scheduled across processors.

The *flat* implementation is obtained by flattening `smvp`. To flatten `smvp` we replace the nested sequence representation of A with a linearized representation (A', s). Here A' is an array of r pairs, indexed by `val` and `col`, partitioned into rows of A by s. Application of the flattening tranformations to the loop in Fig. 2 yields

$$y \texttt{ = segmented\_sum(}A'\texttt{\%val * }x\texttt{(}A'\texttt{\%col),}s\texttt{)},$$

where `segmented_sum` is a data-parallel operation with efficient parallel implementations [3]. By substituting A'`%val * `x`(`A'`%col)` for the first argument in the body of `segmented_sum`, the sum and product may be fused into a *segmented dot-product*. The resulting algorithm was implemented in Fortran 90 for our two target architectures.

For the SGI Origin 200, A' is divided into $p\sigma$ sections of length $r/(p\sigma)$ where p is the number of processors and $\sigma \geq 1$ is a factor to improve the load balance in the presence of multiprogramming and operating system overhead on the processors. Sections are processed independently and dot products are computed sequentially within each section. Sums for segments spanning sections are adjusted after all sections are summed.

For the NEC SX-4, A' is divided into pq sections where q is the vector length required by the vector units [5]. Section i, $0 \leq i < pq$, occupies element $i \bmod q$ in a length q vector of thread $\lfloor i/p \rfloor$. Prefix dot-products are computed independently for all sections using a sequence of $r/(pq)$ vector additions on each processor. Segment dot-products are computed from the prefix dot-products and sums for segments spanning sections are adjusted after all sections are summed [25]. On the SX-4, σ is typically not needed since the operating system performs gang-scheduling and the threads experience very similar progress rates.

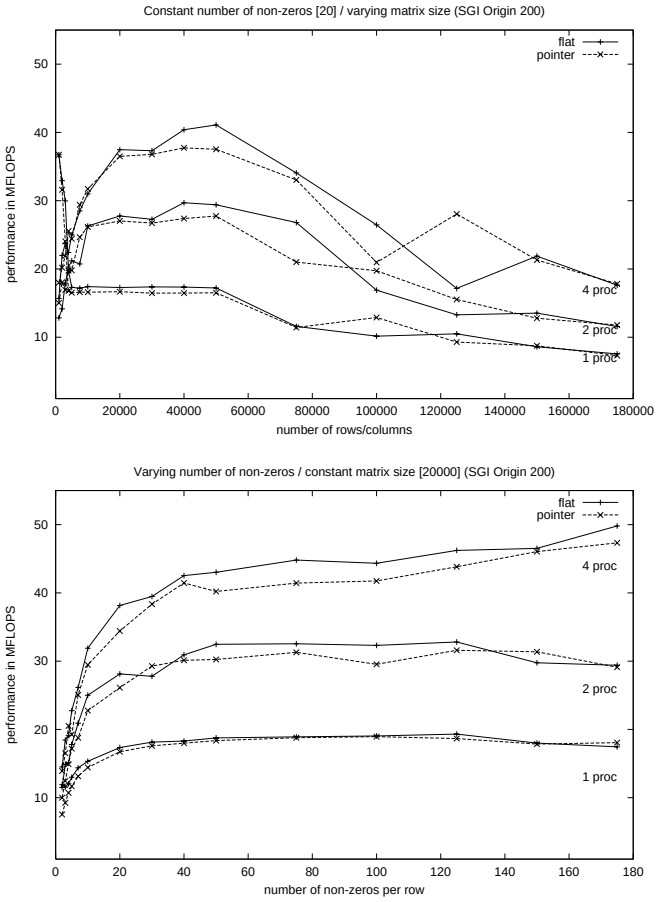


Fig. 4 Performance measurements for the pointer-based and the flattened implementations of `smvp` on the SGI Origin 200.

4.4 Results

The SGI Origin 200 used is a 4 processor cache-based shared memory multiprocessor. The processors are 180MHz R10000 with 1MB L2 cache per processor. The NEC SX-4 used is a 16 processor shared-memory parallel vector processor with vector length 256. Each processor has a vector unit that can perform 8 or 16 memory reads or writes per cycle. The clock rate is 125 MHz. The memory subsystem provides sufficient sustained bandwidth to simultaneously service independent references from all vector units at the maximum rate.

The performance on square sparse matrices of both implementations is shown for 1, 2, and 4 processors for the Origin 200 in Fig. 4 and for the SX-4 in Fig. 5. The top graph of each figure shows the performance as a function of problem size in megaflops

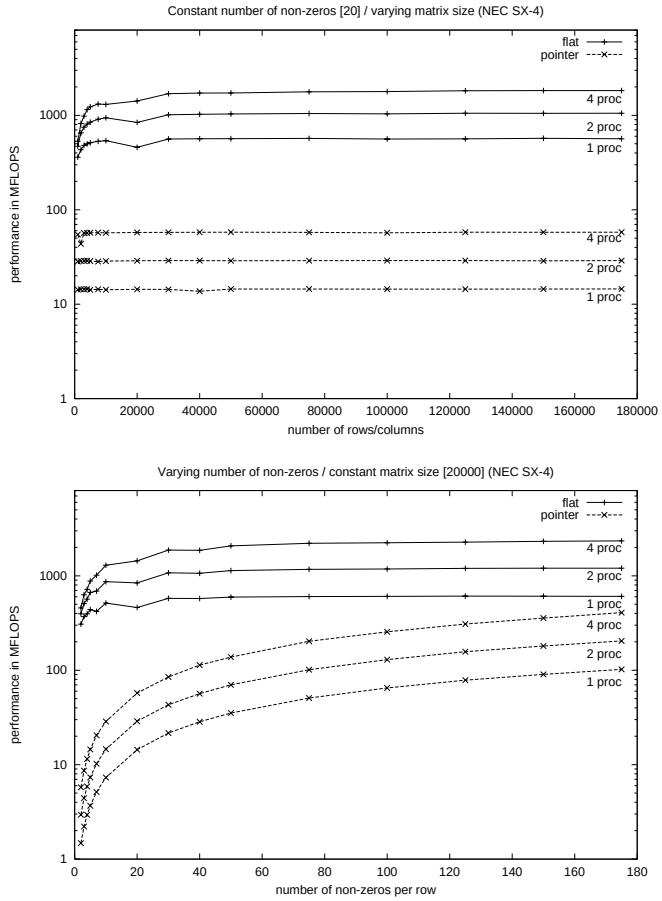
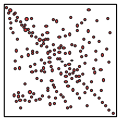


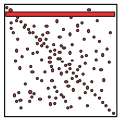
Fig. 5 Performance measurements for the pointer-based and the flattened implementations of `smvp` on the NEC SX-4. (Note the logarithmic scale of the y-axis.)

per second, where the number of floating point operations for the problem is $2r$. Each row contains an average of 20 nonzeros and the number of rows is varied between 1000 and 175000. The bottom graph shows the influence of the average number of nonzeros per row (r/n) on the performance of the code. To measure this, we chose a fixed matrix size ($n = 20000$) and varied the average number of nonzeros on each row between 5 and 175. In each case, the performance reported is averaged over 50 different matrices.

On the Origin 200 the flattened implementation performed at least as well as the pointer-based version over most inputs. The absolute performance of neither implementation is particularly impressive. The sparse matrix-vector problem is particularly tough for processors with limited memory bandwidth since there is no temporal locality in the use of A (within a single matrix-vector product), and the locality in reference to x diminishes with increasing n. While reordering may mitigate these effects in some



(a)



(b)

	SGI Origin 200		NEC SX-4	
	flat	pointer	flat	pointer
regular (a)	41	42	1870	80
irregular (b)	40	26	1278	76

Fig. 6 Performance in Mflops/s using four processors on two different problems.

applications, it has little effect for the random matrices used here. The Origin 200 implementations also do not exhibit good parallel scaling. This is likely a function of limited memory bandwidth that must be shared among the processors. Higher performance can be obtained with further tuning. For example, the current compiler does not perform optimizations to map the `val` and `col` components of A into separate arrays. When applied manually, this optimization increases performance by 25% or more.

On the SX-4 the flattened implementation performs significantly better than the pointer implementation over all inputs. This is because the flattened implementation always operates on full-sized vectors (provided $r \geq pq$), while the pointer-based implementation performs vector operations whose length is determined by the number of nonzeros in a row. Hence the pointer-based implementation is insensitive to problem size but improves with average row length. For the flattened implemementation, absolute performance and parallel scaling are good primarily because the memory system has sufficient bandwidth and the full-sized vector operations fully amortize the memory access latencies.

Next, we examined the performance on two different inputs. The *regular* input is a square sparse matrix with $n = 25000$ rows. Each row has an average of 36 randomly placed nonzeros for a total of $r = 900000$ nonzeros. The *irregular* input is a square sparse matrix with $n = 25000$ rows. Each row has 20 randomly placed nonzeros, but now 20 consecutive rows near the top of A contain 20000 nonzeros each. Thus the total number of nonzeros is again 900000, but in this case nearly half of the work lies in less than 0.1% of the dot products.

The performance of the two implementations is shown in Fig. 6. The pointer-based implementation for the Origin 200 is significantly slower for the irregular problem, regardless of the thread scheduling technique used (dynamic or static). The problem is that a small "bite" of the iteration space may contain a large amount of work, leading to a load imbalance that may not be correctable using a dynamic scheduling technique. In the case of the SX-4 pointer-based implementation this effect is not as noticeable, since the dot product of a dense row operates nearly two orders of magnitude faster than the dot product of a row with few nonzeros.

The flattened implementation delivers essentially the same performance for both problems on the Origin 200. The SX-4 performance in the irregular case is reduced

because dense rows span many successive sections, and incur an $O(pq)$ cost in the final sum adjustment phase that is not present for shorter rows. However, this cost is unrelated to problem size, so the disparity between the performance in the two problems vanishes with increasing problem size.

4.5 Discussion

This example provides some evidence that the flattening technique can be used in an implementation to improve the performance stability over irregular problems while maintaining or improving on the performance of the simple thread-based implementation. The flattening techniques may be particularly helpful in supporting the instruction-level and memory-level parallelism required for high performance in modern processors. The example also illustrates that dynamic thread scheduling techniques, in the simple form generated by Fortran compilers, may not be sufficient to solve load imbalance problems that may arise in irregular nested data-parallel computations.

While these irregular matrices may not be representative of typical problems, the basic characteristic of large amounts of work in small portions of the iteration space is not unusual. For example, it can arise with data structures for the adaptive spatial decomposition of a highly clustered n-body problem, or with divide-and-conquer algorithms like quicksort or quickhull [4].

5 Related work

The facilities for data abstraction and dynamic aggregates are new in Fortran 90. Previously, Norton et al. [21], Deczyk et al. [14], and Nyland et al. [22] have experimented with these advanced features of Fortran 90 to analyze their impact on performance.

HPF 2.0 provides a `MAPPED` irregular distribution to support irregular computations. This is a mechanism, and makes the user responsible for developing a coherent policy for its use. Further, the ramifications of this distribution on compilation are not yet fully resolved. Our approach is fundamentally different in attempting to support well a smaller class of computations with an identifiable policy (nested data parallelism) and by preprocessing the irregular computation to avoid reliance on untested strategies in the HPF compiler. While HPF focuses on the irregular distribution of regular data structures, our approach is based on the (regular) distribution of irregular data structures.

Split-C [13] also provides a number of low-level mechanisms for expressing irregular computations. We are attempting to provide a higher level of abstraction while providing the same level of execution efficiency of low-level models.

The Chaos library [28] is a runtime library based on the inspector/executor model of executing parallel loops involving irregular array references. It is a suitable back end for the features supporting irregular parallelism in HPF 2.0. The library does not provide obvious load balancing policies, particularly for irregularly nested parallel loops. Recent work on Chaos is looking at compilation aspects of irregular parallelism.

Flattening transformations have been implemented for the languages NESL [7], Proteus [26], Amelia [30], and V [11], differing considerably in their completeness and in

the associated constant factors. There has been little work on the transformation of imperative constructs such as sequential loops within a `FORALL`, although there do not appear to be any immediate problems. The flattening techniques are responsible for several hidden successes. Various high performance implementations are really hand-flattened nested data-parallel programs: FMA [15], radix sort [32], as well as the NAS CG implementation described in the introduction. Furthermore, the set of primitives in HPFLIB itself reflects a growing awareness and acceptance of the utility of the flattening techniques.

The mainstream performance programming languages Fortran and SISAL [10, 31] can express nested data parallelism, but currently do not address its efficient execution in a systematic way. Languages that do address this implementation currently have various disadvantages: they are not mainstream languages (NESL, Proteus); they subset or extend existing languages (Amelia, V, F90V); they do not interface well with regular computations (NESL, Proteus); they are not imperative, hence provide no control over memory (NESL, Proteus); and they are not tuned for performance at the level of Fortran (all).

6 Conclusions

Nested data parallelism in Fortran is attractive because Fortran is an established and important language for high-performance parallel scientific computation and has an active community of users. Many of these users, who are now facing the problem of implementing irregular computations on parallel computers, find that threading and flattening techniques may be quite effective and are tediously performing them manually in their codes [22, 15]. At the same time, they have substantial investments in existing code and depend on Fortran or HPF to achieve high performance on the regular portions of their computations. For them it is highly desirable to stay within the Fortran framework.

The advanced features of modern Fortran dialects, such as derived data types, modules, pointers, and the `FORALL` construct, together constitute a sufficient mechanism to express complex irregular computations. This makes it possible to express both irregular and regular computations within a common framework and in a familiar programming style.

How to achieve high performance from such high-level specifications is a more difficult question. The flattening technique can be effective for machines with very high and uniform shared-memory bandwidth, as that found in current parallel vector processors from NEC and SGI/Cray or the parallel multithreaded Tera machine. For cache-based shared-memory processors, the improved locality of the threading approach is a better match. The flattening techniques may help to extract threads from a nested parallel computation that, on the one hand, are sufficiently coarse grain to obtain good locality of reference and amortize scheduling overhead, and, on the other hand, are sufficiently numerous and regular in size to admit good load balance with run-time scheduling.

Thus we believe that irregular computations can be expressed in modern Fortran dialects and efficiently executed through a combination of source-to-source preprocessing, leveraging of the Fortran compilers, and runtime support. Looking ahead, we are

planning to examine more complex irregular algorithms such as supernodal Cholesky factorization, and adaptive fast n-body methods.

References

1. P. Au, M. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, G. Keller, M. Köhler, M. Simons, and W. Pfannenstiel. Enlarging the scope of vector-based computations: Extending Fortran 90 with nested data parallelism. In W. Giloi, editor, *International Conference on Advances in Parallel and Distributed Computing*, pages 66–73. IEEE, 1997.
2. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
3. G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, Cambridge, MA, 1990.
4. G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, Mar. 1996.
5. G. E. Blelloch, S. Chatterjee, and M. Zagha. Solving linear recurrences with loop raking. *Journal of Parallel and Distributed Computing*, 25(1):91–97, Feb. 1995.
6. G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, CA, June 1995.
7. G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.
8. G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, Feb. 1990.
9. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995. ACM.
10. D. Cann and J. Feo. SISAL versus FORTRAN: A comparison using the livermore loops. In *Proceedings of Supercomputing'90*, pages 626–636, New York, NY, Nov. 1990.
11. M. M. T. Chakravarty, F.-W. Schröer, and M. Simons. V—Nested parallelism in C. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models for Massively Parallel Computers*, pages 167–174. IEEE Computer Society, 1995.
12. S. Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Trans. Prog. Lang. Syst.*, 15(3):400–462, July 1993.
13. D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Nov. 1993.
14. V. K. Decyk, C. D. Norton, and B. K. Szymanski. High performance object-oriented programming in Fortran 90. *ACM Fortran Forum*, 16(1), Apr. 1997.
15. Y. C. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997. ACM.
16. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. The MIT Press, Cambridge, MA, 1994.

17. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sept. 1979.
18. J. McCalpin. Personal communication, Apr. 1998.
19. M. Metcalf and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996.
20. G. J. Narlikar and G. E. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, Las Vegas, NV, June 1997. ACM.
21. C. D. Norton, B. K. Szymanski, and V. K. Decyk. Object-oriented parallel computation for plasma simulation. *Commun. ACM*, 38(10):88–100, Oct. 1995.
22. L. S. Nyland, S. Chatterjee, and J. F. Prins. Parallel solutions to irregular problems using HPF. First HPF UG meeting, Santa Fe, NM, Feb. 1997.
23. OpenMP Group. OpenMP: A proposed standard API for shared memory programming. White paper, OpenMP Architecture Review Board, Oct. 1997.
24. D. W. Palmer, J. F. Prins, S. Chatterjee, and R. E. Faith. Piecewise execution of nested data-parallel programs. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 346–361. Springer-Verlag, 1996.
25. J. Prins, M. Ballabio, M. Boverat, M. Hodous, and D. Maric. Fast primitives for irregular computations on the NEC SX-4. *Crosscuts*, 6(4):6–10, 1997. CSCS.
26. J. F. Prins and D. W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA, May 1993.
27. S. Saini and D. Bailey. NAS Parallel Benchmark (1.0) results 1-96. Technical Report NAS-96-018, NASA Ames Research Center, Moffett Field, CA, Nov. 1996.
28. J. Saltz, R. Ponnusammy, S. Sharma, B. Moon, Y.-S. Hwang, M. Uysal, and R. Das. A manual for the CHAOS runtime library. Technical Report CS-TR-3437, Department of Computer Science, University of Maryland, College Park, MD, Mar. 1995.
29. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, NY, 1986.
30. T. J. Sheffler and S. Chatterjee. An object-oriented approach to nested data parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 203–210, McLean, VA, Feb. 1995.
31. S. K. Skedzelewski. Sisal. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–157. ACM Press, New York, NY, 1991.
32. M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of Supercomputing'91*, pages 712–721, Albuquerque, NM, Nov. 1991.