To appear: *Crosscuts* **6** (4) Dec 1997 (`http://www.cscs.ch/Official/PubCrosscuts6-4.pdf`)

Fast Primitives for Irregular Computations on the NEC SX-4

J.F. Prins[†], University of North Carolina at Chapel Hill, USA

M. Ballabio, M. Boverat, M. Hodous, D. Maric, CSCS/SCSC Swiss Center for Scientific Computing, Switzerland

1. IRREGULAR COMPUTATION

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 easily become prohibitive. Fortunately, sophisticated modeling techniques have been developed in many areas that that vary model resolution as needed coupled with sparse and adaptive algorithms that vary computational effort in time and space as needed. For example, adaptive spatial decompositions are used with fast n-body techniques such as FMA or P^3M in cosmological simulations, and adaptive unstructured meshing and sparse linear system solvers are used in computational fluid dynamics. The computations that result from these techniques are called *irregular*, reflecting the *a priori* unpredictability in the reference pattern and distribution of work.

Irregular computations can be problematic for current cache-oriented parallel computers (NUMA), where high performance requires equal distribution of work over processors as well as locality of reference within each processor. For many irregular computations load balance and locality trade in a complex fashion, leading to complex optimization problems in problem decomposition that are input-dependent and may evolve with the computation itself.

Parallel computers that provide uniform access to a shared memory (UMA) do not require locality of reference, and can optimize irregular computations entirely for load balance, which can be performed quite effectively at run-time. However, high-performance UMA architectures require memory references to be presented in bulk by each processor in order to amortize access latency. In parallel computers such as the NEC SX-4, this is achieved through the use of vector operations within each processor.

Manual or automated techniques can be used to restructure a large class of irregular computations to express them in terms of simple vector operations and a small set of primitive operations. The performance of irregular parallel computations expressed in this fashion depends largely on the parallel performance of these primitives. Here we briefly describe the implementation of one such primitive on the NEC SX-4 and illustrate its use in achieving record performance on the NPB CG irregular computation benchmark.

2. EXPRESSING IRREGULAR COMPUTATIONS IN FORTRAN

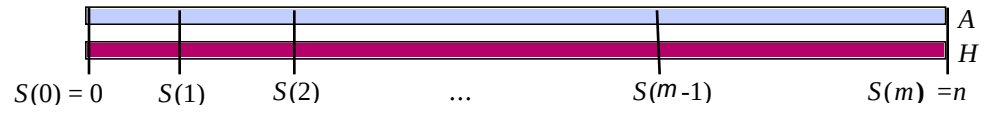
Irregular computations are difficult to express using performance-oriented languages such as Fortran, because there is a basic mismatch between the rectangular arrays of

---

[†] Work performed at the Institute for Theoretical Computer Science, IFW E48.2, ETH Zürich.

Fortran and the data types that are characteristic of irregular computations, such as trees, graphs and nested sequences. While irregular data types can always be represented using simple 1-D arrays, doing so results in programs that are beyond compile-time analysis. Compilers have difficulty generating high-performance parallel code for such programs.

Consider, for example, the product of a sparse matrix $M^{m \times m}$ with a vector V^{m}, where M has n nonzeros. A *compressed row representation* for M uses two data arrays $A(0{:}n\text{-}1)$ and $H(0{:}n\text{-}1)$, representing, respectively, the non-zero values in consecutive rows of and the columns in which they occur. In addition, a separate array $S(0{:}m)$ describes the partitioning of A and H into rows of M, by recording the positions at which successive rows of M begin.



With such a representation the product $R = M \bullet V$ can be expressed in Fortran 77 with the following nested loops:

```
DO i = 0, m-1
     R(i) = 0
     DO j = S(i), S(i+1)-1
          R(i) = R(i) + A(j) * V( H(j) )
     ENDDO
ENDDO
```

In a Fortran 77 program such as the one above, the available parallelism must be discovered and extracted from the sequential DO loops. This is not always a simple matter, and therefore more modern Fortran dialects such as Fortran 90, Fortran 95 and High Performance Fortran (HPF) provide array-parallel operations and the FORALL construct to specify available parallelism explicitly

Whether through nested loops or nested parallel constructs, our example program specifies *nested data-parallelism* in which the actual degree of parallelism is data-dependent. This kind of parallelism is complex to implement. Consider the options for the Fortran 77 program above.

(a) We may parallelize the inner loop, in essence creating a parallel dot product. However, this will give poor performance if M has rows with few nonzeros. The average number of nonzeros per row is quite small in most sparse linear systems.

(b) We may parallelize the outer loop, performing a number of sequential dot products in parallel. This approach leads to load imbalance if we have rows that vary widely in the number of nonzeros, because some rows will take longer to complete than others.

Neither approach in itself, nor both together, is adequate. A better approach arranges the parallel work to be independent of the structure of M. Conceptually we flatten the iteration space of the nested loops into a single-dimensional iteration space, replacing

reduction operations, such as the summation, with a primitive that performs multiple reductions in parallel. In this case we obtain the following Fortran 90 program:

```
T = A * V(H)
R = SEGMENTED_SUM_REDUCE(T, S)
```

The computation of T uses simple array-parallel operations and is independent of the actual structure of A and H. `SEGMENTED_SUM_REDUCE`(T,S) is defined as follows: given $T(0{:}\ n\text{-}1)$ and a segmentation descriptor $S(0{:}\ m)$, the result $R(0{:}\ m{-}1)$ satisfies $0 \le i < m$ $R_i = \mathrm{SUM}(T(S_i{:}\ S_{i+1}{-}1))$. Provided this primitive can be implemented to give good parallel performance regardless of the size distribution of the segments, the flattened program (a) retains all the parallelism specified in the original program and (b) is not dependent on the structure of sparse matrix.

The mechanical flattening of nested data parallelism has been studied in high-level functional languages such as NESL and Proteus [BCH+94, PP93]. Its application in imperative languages such as Fortran is the subject of current research. Irregular applications may also be constructed directly using F90/HPF and the primitives in Figure 1 [NCP97, HTJ97]. Examples include adaptive fast-multipole n-body methods [HJ96] and graph partitioners [HJT97].

While expressing irregular computations in Fortran is generally more complex than it would be using NESL or Proteus, the use of Fortran simplifies the interface between regular and irregular portions of a computation and leverages optimizing Fortran compiler technology to provide a seamless interface between the regular and flattened irregular portions of the computation.

3. PRIMITIVES FOR IRREGULAR COMPUTATION

Segmented sum is one of the primitives for irregular computation shown in Figure 1. Effective implementation of these operations must have excellent parallel scalability, and high absolute performance that is parametric in the overall work performed and is largely insensitive to the data values (e.g. sizes of segments). This property is fundamental for the construction of irregular applications with predictable performance.

- Eliminate or insert elements in sequences
  `PACK, UNPACK, MERGE`
- perform multiple independently-sized combining operations
  `SEGMENTED_XXX_REDUCE, XXX_SCATTER`
- compute data rearrangements
  `SEGMENTED_XXX_PREFIX, GRADE_UP, GRADE_DOWN`

Figure 1. Primitives for irregular computation. `XXX` stands for a combining operation such as `SUM`, `PRODUCT`, `AND`, `OR`, etc.

The primitives are closely related to the HPFLIB intrinsics, and could be defined using HPFLIB. In practice, the commercial implementations of the HPFLIB primitives in the systems we have seen fail to meet the performance requirements above [NCP97]. In some cases the operations fail to parallelize, and in other cases the implementations exhibit performance that is highly dependent on the input data. One reason for this situation may

be that the algorithms and analysis for optimum performance are complex. This project is part of a comprehensive algorithm design, analysis, and implementation effort to address this problem.

4. IMPLEMENTATION OF SEGMENTED SUM

Let p be the number of processors and let q be the number of elements required for a bulk reference. For a vector processor q is the length of a vector register. We treat the input sequence T as an (h, pq) array in column-major order where $h = \lceil n/pq \rceil$ as shown in Figure 2. Segment ends are shown as breaks in the column, but in actuality all data in T is contiguous.

For each column a sequential summation computes successive prefix sums into a second array U. All columns can be summed simultaneously by assigning each column to a unique element of a vector register in a processor and letting each processor repeatedly perform a vector addition from T and a vector store to U. Since each column is the same length, we get very precise load balance.

To extract the results, let W be the final column sums and define $R(i) = U(S(i+1)) - U(S(i))$. R differs from the correct segment sum in at most pq positions where a segment spans one or more columns (shown by the shaded segment ends). The correction value for these partial sums in R is given by the segmented sum of W where the W segments are defined by the columns in which a T segment ends.

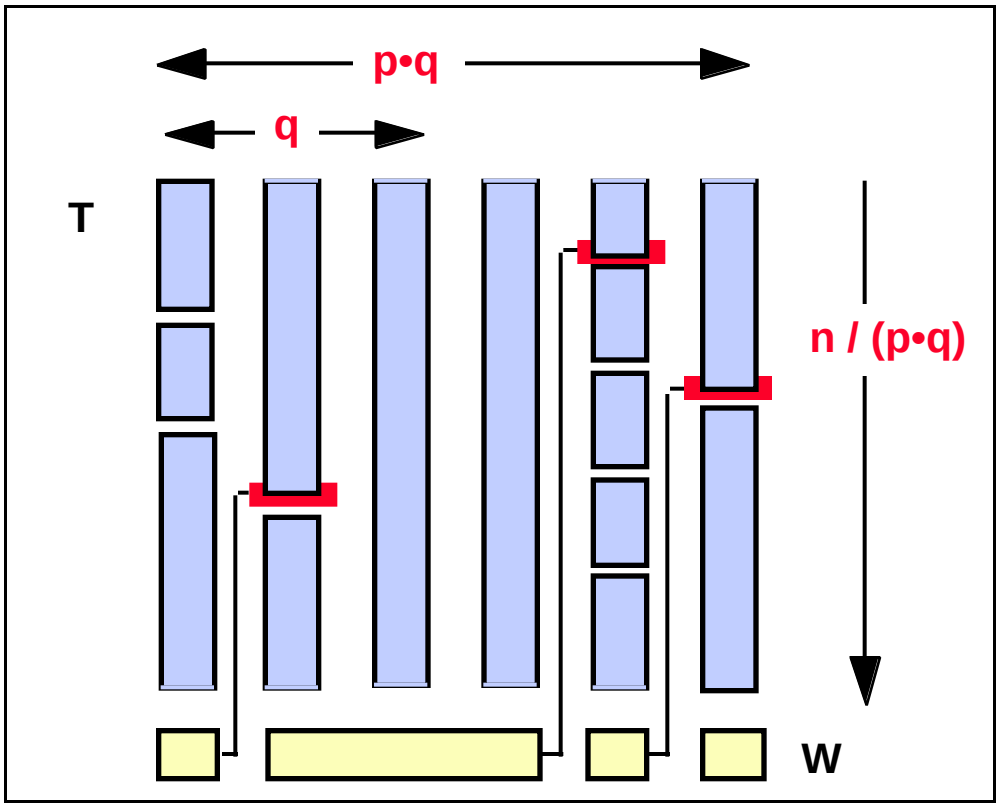


Figure 2: Implementation of segmented sum

The performance of implementations of segmented sum on the NEC SX-4 and the Cray T90 are shown in Figure 3. The performance is compared against *nested sum*, which is the result of a standard compilation of the nested loop implementation of segmented sum. That implementation is quite sensitive to the average segment size because the inner loop is vectorized while the outer loop is parallelized (single processor performance is shown here); there is a factor of 500 performance difference over the spectrum of inputs. In contrast, the segmented sum implementation on the SX-4 operates at 94% of peak performance of the machine, and the ratio of longest running time to shortest running time is about 4.

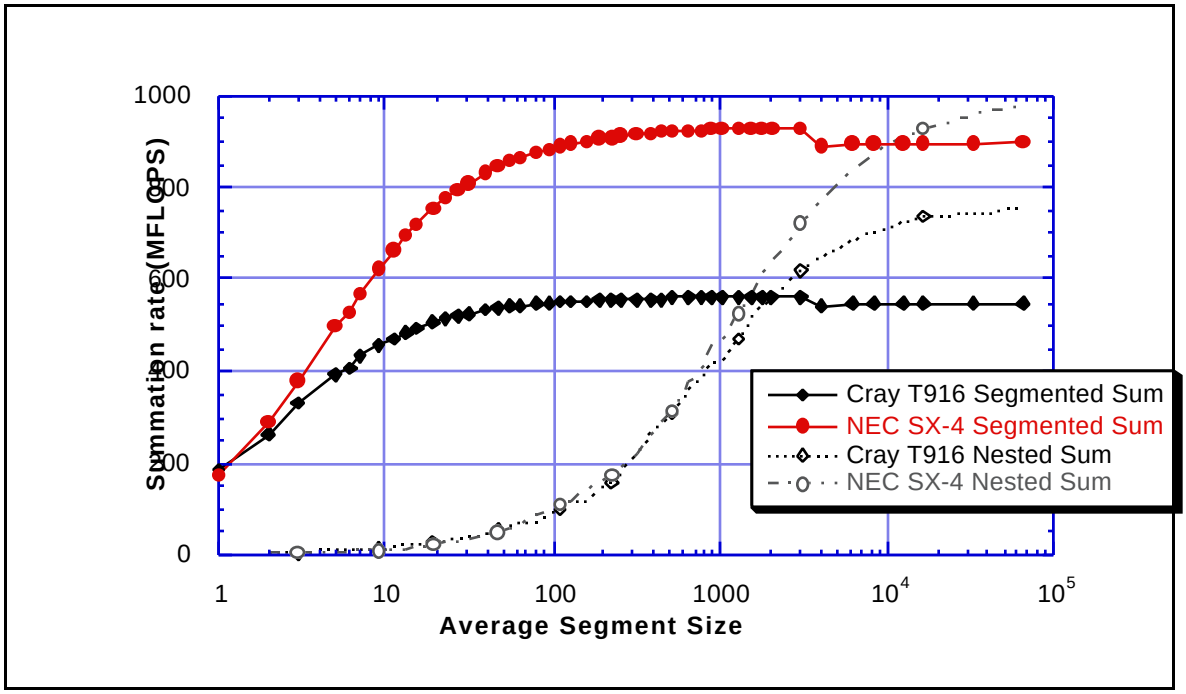


Figure 3: Performance of segmented sum implementations.

5. APPLICATION: NPB CG BENCHMARK

To give a simple illustration of the use of the segmented sum primitive, we replaced the sparse matrix-vector product in the distributed sample code for the NAS NPB 1.0 CG benchmark with the flattened program shown in section 2. We also added some compiler directives to parallelize and vectorize the remaining regular vector-vector operations. NPB 1.0 is a "best effort" benchmark, in which implementations may be optimized in this fashion.

Figure 4 plots the performance of the CG implementations on the class B benchmark. The highest reported performance at the time of this writing is 5.1 GFLOPS for a 16 processor Cray C90 [SB96]. The highest NUMA performance documented is 2.5 GFLOPS from a 256 processor Cray T3E. Our implementation for the SX-4 runs at 8.8 GFLOPS using 16 processors and at 13.3 GFLOPS using 32 processors, and hence achieves world-record performance on this problem.

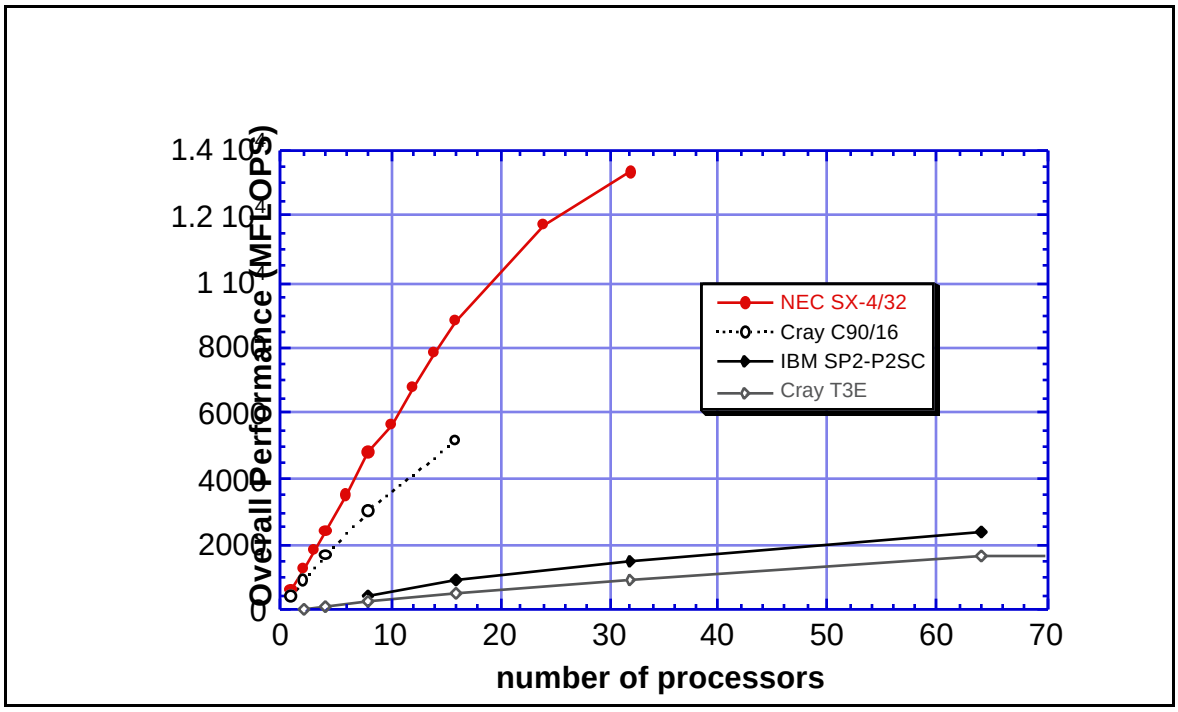


Figure 4: NAS NPB 1.0 CG performance. The NEC implementation uses the sparse matrix - vector product described in section 2.

6. CONCLUSIONS AND FURTHER WORK

We have described a technique to implement irregular computations by flattening nested data parallelism and supported by a small library of primitives. We believe this approach significantly simplifies the construction of complex irregular algorithms and can offer predictable performance in the face of varying inputs. The implementation of the primitives is the key determinant of this property. On high-performance UMA architectures the primitives may be implemented to achieve high performance and load balance by using a factor q of "parallel slack" to meet the bulk-reference condition.

UMA architectures are currently only found in supercomputers like the NEC SX-4 and Cray T90 which are quite expensive relative to NUMA architectures constructed using commodity processors. However, as the CG results show, for some important irregular computations these machines offer the highest performance available at *any* price. In the near term it is possible that the design of parallel computers constructed from commodity processors will also be driven towards bulk-reference operation, through vector operations, multithreading, or user-specified caching policies implemented through bulk rearrangement of data. One impetus comes from the increasing shortfall of memory performance for increasingly fast processors; another comes from the vacuum created by the decreasing numbers of conventional supercomputers.

While segmented sum (and other segmented reductions) are one of the most important primitives for irregular computation, there are more to implement. In our project we are now considering implementations for other operations from the list in Figure 1.

DEDICATION

We dedicate this project in memory of Matteo Boverat who died unexpectedly while this project was underway. Matteo was a major contributor to this effort whose boundless enthusiasm and enjoyment of the work was an inspiration to us all.

REFERENCES

[BCH+94] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, M. Zagha, "Implementation of a Portable Nested Data-Parallel Language", *Journal of Parallel and Distributed Computing* **21** (2), 1994.

[BBB+91] D. Bailey et al., "The NAS Parallel Benchmarks", *Intnl Journal of Supercomputer Applications* **5** (3), 1991.

[HJ96] Y. Hu and L. Johnsson. Implementing O(N) N-body Algorithms Efficiently in Data-Parallel Languages, *Journal of Scientific Programming*, **5** (4), 1996.

[HJT97] Y. Hu, L. Johnsson and S.-H. Teng. High Performance Fortran for Highly Unstructured Problems. *Proc. 6th Symposium on Principles and Practice of Parallel Programming*, ACM, 1997.

[HTJ97] Y. Hu, S.-H. Teng and L. Johnsson, "A Data-Parallel Implementation of the Geometric Partitioning Algorithm", Proceedings of the *8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[NCP97] L. Nyland, S. Chatterjee, J. Prins, "Parallel Solutions to Irregular Problems Using HPF", First HPF UG meeting, Santa Fe, NM, 1997.

[PP93] J. Prins, D. Palmer, "Transforming High-level Data-Parallel Programs into Vector Operations", *Proc. 5th Symposium on Principles and Practice of Parallel Programming*, ACM, 1993.

[SB96] S. Sani, D. Bailey, "NAS Parallel Benchmark (1.0) Results 11-96", NASA Ames Research Center Technical Report NAS-96-018, Nov. 1996.