Flexible Data Distribution in PGHPF

Mark Leair, Douglas Miles, Vincent Schuster, and Michael Wolfe

The Portland Group, Inc., USA, http://www.pgroup.com

Abstract. We have implemented the GEN_BLOCK (generalized block) data distribution in PGHPF, our High Performance Fortran implementation. Compared to a BLOCK or CYCLIC distribution, the more flexible GEN_BLOCK distribution allows users to balance the load between processors. Simple benchmark programs demonstrate the benefits of the new distribution format for unbalanced work loads, getting speedup of up to 2X over simple distributions. We also show performance results for a whole application.

High Performance Fortran (HPF) gives programmers a simple mechanism to run parallel applications by distributing arrays across multiple processors. HPF programs are single-threaded, achieving data parallelism through parallel loops and array assignments. The performance of HPF programs depends to a great deal on the data distribution, and on the optimizations in the compiler to recognize and reduce communications.

HPF compilers are responsible for implementing the distributions and generating any necessary synchronization or communication among the processors. In HPF-1, the two main distribution formats were BLOCK and CYCLIC. In a BLOCK distribution, the size of the array was divided as evenly as possible across all processors, with each processor getting a contiguous chunk of the array. In a CYCLIC distribution, the elements of the array are distributed round-robin. The choice of which distribution to use is up to the programmer, and depends on communication patterns and load balancing issues; in both cases, each processor gets roughly equal parts of the array.

HPF-2 introduced a new *generalized BLOCK* or GEN_BLOCK distribution format; a GEN_BLOCK distributed array can have different sized blocks on each processor; the block size for each processor is under user control, allowing flexible load balancing.

At the Portland Group, Inc. (PGI), we have implemented the GEN_BLOCK distribution format in our PGHPF compiler. We report on the changes required to support the new distribution format, and give some performance results.

1 The GEN_BLOCK Distribution Format

Suppose we have a 100×100 matrix `A` where we distribute the rows across four processors using a BLOCK distribution; each processor then "owns" 25 rows of the matrix. This is specified in HPF with the directive

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 418–421, 1999.

```
!hpf$ distribute A(block,*)
```

If the amount of work in the program is the same for each row, this distribution should balance the load between processors. If we are only operating on the lower triangle of the matrix, processor zero operates on 325 elements, while processor three operates on 2200, a very unbalanced workload.

One way to try to balance the workload in a simple example like this is to use a CYCLIC distribution, but this often adds communication overhead as well as additional subscript calculation. Block-cyclic (CYCLIC(N)) distributions add significant communication, index computation, and loop overhead, and its use is discouraged in most HPF compilers. A GEN_BLOCK distribution allows us to retain the advantages of a BLOCK distribution, while distributing the rows in a fashion to balance the workload across processors. If we allocate the rows to processors as shown in the following table, the workload is balanced within 5% across processors:

proc	rows	work
0	1:50	1275
1	51:71	1281
2	72:87	1272
3	88:100	1222

This could be specified in HPF with the statement

```
!hpf$ distribute matrix(gen_block(gb))
```

where `gb` is an integer vector with values $\{50, 21, 16, 13\}$. The GEN_BLOCK vector can be computed at runtime, as long as its values are computed before the matrix is allocated.

Many Fortran applications have irregular data structures implemented using vectors and arrays, where a simple BLOCK distribution causes unnecessary load imbalance and communication. A GEN_BLOCK distribution will improve performance in many of these applications.

2 Implementing GEN_BLOCK in PGHPF

We recently added support for GEN_BLOCK in our PGHPF compiler. Since GEN_BLOCK is so close to BLOCK, the changes in the compiler itself were straightforward. Both are implemented with a local dynamically allocated array on each processor and a local descriptor containing information about the global distribution and the local subarray. Array assignments and parallel loops on the BLOCK or GEN_BLOCK arrays are localized into a single loop over the local subarray. We support *shadow* regions for BLOCK distributions; this was easy to carry forward to GEN_BLOCK distributions as well. When the compiler recognizes nearest-neighbor communication, it issues a single collective communication to exchange shadow regions before entering the computation loop; computation can then proceed without communication.

The compiler must manage the GEN_BLOCK distribution vector (`gb` in our example above); since the distribution vector might be changed after the distributed array is allocated, the compiler generates code to copy the vector to a temporary variable.

One major difference between a GEN_BLOCK distribution and a BLOCK distribution is for random indexing. Given a subscript value i in a BLOCK-distributed dimension the compiler can generate code to determine the processor index that owns i by computing $\lfloor i/b \rfloor$, where b is the block size. In a GEN_BLOCK distributed dimension, there is no simple formula to determine the processor that owns subscript i; the fastest code involves a binary search using the GEN_BLOCK distribution vector, which we implement in a library routine. While this is potentially very costly, most common communication patterns are structured, avoiding the search.

Most of the implementation effort went into augmenting the interprocessor communication library to support communication between GEN_BLOCK and BLOCK or CYCLIC arrays, and between different GEN_BLOCK distributed arrays. There library incurs additional overhead to consult the GEN_BLOCK distribution vector, but in irregular problems, the benefits of load balancing more than covers this cost.

3 Performance Results

We first tested our GEN_BLOCK on a simple triangular matrix operation:

```
integer, dimension(np) :: gb
real, dimension(n,n) :: a,b,c
!hpf$ distribute a(gen_block(gb),*)
!hpf$ align with a(i,*) :: b(i,*), c(i,*)
!... initialize a, b, c
do i = 1,n
  do j = 1,i
    a(i,j) = b(i,j) + c(i,j)
  enddo
enddo
```

We initialized the GEN_BLOCK distribution vector `gb` to balance the load among processors, and compared the resulting performance to that of a simple BLOCK distribution. The following table compares execution time of this loop on 1–32 processors and the two distributions on a Hewlett-Packard Exemplar, an IBM SP-2, and a Cray T3E (we were unable to get time on the Cray for 16 or 32 processors). The matrix sizes are given, and the loop was repeated 100 times.

		Time in Seconds					
	processors	1	2	4	8	16	32
HP Exemplar	gen_block	519	126	118	36	44.3	5.4
n=4099	block	524	327	219	146	62.2	46.9
IBM SP-2	gen_block	407	178	81	36.8	17.1	8.2
n=2053	block	407	240	119	49.8	23.2	13.8
Cray T3E	gen_block	173	87	44.7	22.1		
n=2053	block	172	136	77.0	40.9		

As an example of how GEN_BLOCK distribution can help balance the load, the following table shows the time for each of 8 processors on the Cray T3E with BLOCK and GEN_BLOCK distributions, for 100 iterations the benchmark loop with matrix size 2053×2053:

	P0	P1	P2	P3	P4	P5	P6	P7
gen_block	10.8	18.8	20.4	21.1	21.6	21.6	21.9	22.1
block	0.72	4.11	10.2	16.6	22.9	29.0	35.3	40.9

Since GEN_BLOCK is new, few applications have been modified to take advantage of it. We were able to find customer program implementing an adaptive, hierarchical, N-body particle-in-cell code (Hu et al. 1997) that had been modified in anticipation of the availability of GEN_BLOCK. We executed this program with a test input on 32 processors of an IBM SP-2, and compared the performance of GEN_BLOCK distribution to a BLOCK distribution, with time in milliseconds:

	processors	time
block	32	6562
gen_block	32	5000
improvement		1.31

In this application, the GEN_BLOCK distributed version completed 31% faster than the BLOCK version; we note the GEN_BLOCK distribution vector was carefully computed by the author to balance the load.

In conclusion, the compiler modifications to support the GEN_BLOCK distribution were relatively minor; most of the effort was in the communications library. With GEN_BLOCK, most irregular HPF applications can now balance their own workload across processors.

References

[HPF] High Performance Fortran Language Specification, Version 2.0 (1997)

[Hu] Hu, Y. C., Johnnson, S. L., Teng, S-H., High Performance Fortran for Highly Irregular Problems, Proc. Sixth Symp. Principles and Practices of Parallel Programming, ACM Press (1997) 13–24