Optimizing the NPB CG benchmark for multi-core AMD Opteron microprocessors

Stephen Whalen
Cray, Inc.

August 29, 2007

1 Description of CG

1.1 High-level description

CG approximates the largest eigenvalue of a sparse, symmetric, positive definite matrix, using inverse iteration [3]. The matrix is generated by summing outer products of sparse vectors, with a fixed number of nonzero elements in each generating vector. The matrix sizes and total number of nonzero elements ("computed nonzeros," following [3]) are listed in Table 1. The benchmark computes a given number of eigenvalue estimates, referred to as "outer iterations," using 25 iterations of the conjugate gradient method to solve the linear system in each outer iteration.

NPB Class	Global matrix size	Computed nonzeros	Outer iterations
B	$75{,}000^2$	13,708,072	75
D	$1{,}500{,}000^2$	694,392,620	100

Table 1: Problem parameters for NPB CG

1.2 Implementational details

1.2.1 The serial benchmark

The dominant computation in each iteration in the inverse iteration algorithm is the solution of a linear system. As mentioned above, NPB CG takes its name from the conjugate gradient algorithm used to approximately solve this system. In turn, each iteration of the conjugate gradient algorithm is computationally dominated by a matrix-vector multiplication.

The reference implementations generate the sparse matrix in compressed sparse row (CSR) format, also called AIJ format. This storage scheme uses three arrays to describe a matrix A:

- the array `a(:)` holds only the nonzeros entries of A, in row-major order;

- the array `rowstr(:)` holds the starting indices of the rows in `a`—that is, `a(rowstr(i))` is the first nonzero entry in row `i`;
- the array `colidx(:)` holds the column indices corresponding to the entries of `a`.

The resulting code for the matrix-vector multiplication Ap is shown in Figure 1, annotated with the line numbers where this code appears in `cg.f`. We will see in Section 1.3 that this kernel is indeed the "hotspot" for this benchmark.

```
570:         do j=1,lastrow-firstrow+1
571:            sum = 0.d0
572:            do k=rowstr(j),rowstr(j+1)-1
573:               sum = sum + a(k)*p(colidx(k))
574:            enddo
575:            w(j) = sum
576:         enddo
```

Figure 1: Sparse matrix-vector multiply kernel. The hotspot is the inner loop over `k`.

1.2.2 The MPI benchmark

To this author's knowledge, the NAS reference MPI implementation of CG is not documented in any NAS reports; hence, this subsection lacks citations.

The reference MPI implementation uses a standard two-dimensional block decomposition to distribute the data among the processes. The global matrix is decomposed into blocks, laid out into r process rows and c process columns. The reference code implements its own binary-tree global reduction scheme, which requires that the number of process rows and columns columns both be powers of two. Therefore, the total process count must be a power of two, so that we may assume either $c = r$ or else $c = 2r$. This results in the local matrix sizes given in Table 2.

Vectors are decomposed into c blocks and distributed over the process columns, with each block replicated throughout each process column. Thus, a matrix A and vector x decompose as

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1c} \\ A_{21} & A_{22} & \cdots & A_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ A_{r1} & A_{r2} & \cdots & A_{rc} \end{bmatrix}, \qquad x = (\, x_1 \mid x_2 \mid \cdots \mid x_c \,),$$

NPB Class	**MPI processes**	**Local matrix size**	**Computed nonzeros**
			per-process average
B	2 (packed)	$75{,}000^2$	13,708,072
D	64	$187{,}500^2$	10,849,885
	256	$93{,}750^2$	2,712,471

Table 2: Per-process array sizes for decomposed CG

where, assuming appropriate divisibility of the global problem size n, each A_{ij} is $(n/r) \times (n/c)$, and each x_i has length n/c.

Matrix-vector multiplication proceeds according to the following algorithm.

1. Assign A_{ij} to process (i, j).
2. Assign a copy of x_j to each of processes (k, j), $1 \le k \le r$.
3. Compute $A_{ij}x_j$ on process (i, j).
4. Sum the resulting $A_{ij}x_j$ across process rows; that is, each process (i, k), $1 \le k \le c$, now holds $y_i = \sum_{\ell=1}^{c} A_{i\ell}x_\ell$. At this point, x is distributed over the process columns, while y is distributed over the process rows.
5. Perform a global transpose to distribute y over the process columns. If the number of process rows equals the number of process columns, then process (i, j) sends its copy of y_i to process (j, i), where it replaces the local y_j. For a non-square distribution, the transposition is more complicated, with each process sending and receiving portions of their local y blocks.
6. Process (i, j) now holds corresponding blocks of x and $y = Ax$.

Dot products $x^T y$ (and thus norms, as well) use the following algorithm.

1. Assign copies of x_j and y_j to each of processes (k, j), $1 \le k \le r$.
2. Compute $x_j^T y_j$ on process (i, j).
3. Sum the resulting scalars across process rows.

This reduction, admittedly, replicates work among the process rows. For machines on which communication is much more expensive than computation, this certainly could be an efficient use of resources.

1.3 Profiles

Portions of function-level sampling profiles are shown in Tables 3, 4, and 5. The `conj_grad` subroutine accounts for the majority of the run time in all cases.

Table 3(a) contains the entire profile for a Class B serial run, while Table 3(b) shows the `conj_grad` line-level profile from the same run. As expected, the line-level profile shows that the sparse matrix-vector multiply kernel, shown in Figure 1, is the bottleneck for this benchmark.

2 Optimizations

2.1 Software prefetching

Initially, we shall only look at serial optimizations, to improve the performance of the sparse matrix-vector multiplication kernel.

Samp %	Cum. Samp %	Samp	Imb. Samp	Imb. Samp %	Function
100.0%	100.0%	111354	--	--	Total
98.8%	98.8%	110044	23.00	0.1%	conj_grad_
0.7%	99.5%	731	5.50	3.0%	sparse_
0.2%	99.7%	206	3.00	5.7%	__c_mcopy8
0.1%	99.8%	139	1.50	4.2%	makea_
0.1%	99.9%	112	3.00	10.2%	__c_mzero8

(a) Function-level profile

Samp %	Samp	Imb. Samp	Imb. Samp %	Group Function Source Line
100.0%	111354	--	--	Total
98.8%	110044	--	--	conj_grad_ cg.f
94.2%	103638	35.00	0.1%	line.572
3.8%	4154	6.00	0.6%	line.692
0.7%	719	3.50	1.9%	line.570
0.6%	664	9.00	5.3%	line.652
0.3%	296	8.00	10.3%	line.676
0.3%	280	1.00	1.4%	line.631
0.2%	183	6.50	13.3%	line.662

(b) Line-level profile for `conj_grad`

Table 3: CrayPat sampling profile for CG Class B serial

Samp %	Cum. Samp %	Samp	Imb. Samp	Imb. Samp %	Function
100.0%	100.0%	12086166	--	--	Total
92.4%	92.4%	11164630	1348.66	0.8%	conj_grad_
3.6%	96.0%	437008	461.75	6.4%	PtlEQPeek
0.9%	96.9%	108518	166.41	9.1%	PtlEQGet
0.6%	97.5%	75491	118.45	9.3%	PtlEQGet_internal
0.4%	97.9%	48928	40.50	5.1%	sprnvc_
0.4%	98.3%	43849	85.86	11.3%	ptl_hndl2nal
0.3%	98.6%	41094	67.91	9.7%	sparse_
0.3%	98.9%	37344	80.50	12.3%	fast_nal_poll
0.3%	99.2%	35414	73.66	11.9%	check_eqs_for_event
0.2%	99.4%	24223	43.52	10.5%	randlc_

Table 4: CrayPat sampling profile for CG Class D, 64 processes

Samp %	Cum. Samp %	Samp	Imb. Samp	Imb. Samp %	Function
100.0%	100.0%	9091448	--	--	Total
70.3%	70.3%	6390252	2519.08	9.2%	conj_grad_
14.1%	84.4%	1282126	558.70	10.1%	PtlEQPeek
3.5%	87.9%	317760	205.75	14.3%	PtlEQGet
2.4%	90.3%	220638	153.13	15.1%	PtlEQGet_internal
2.1%	92.4%	193532	53.02	6.6%	sprnvc_
1.4%	93.9%	130182	103.48	17.0%	ptl_hndl2nal
1.2%	95.1%	110365	93.89	18.0%	fast_nal_poll
1.1%	96.2%	104363	96.33	19.2%	check_eqs_for_event
1.0%	97.3%	95280	47.81	11.4%	randlc_
1.0%	98.2%	86981	239.23	41.5%	ioctl

Table 5: CrayPat sampling profile for CG Class D, 256 processes

The inner loop of this kernel makes stride-1 accesses through the arrays `a` and `colidx`, with no reuse, and makes random accesses to `p`. Streaming accesses such as those for `a` and `colidx` will benefit from prefetching, either in hardware or software [1, 2, 4].

In many cases, compilers are able to recognize such access patterns, but PGI's current compilers are unable to automatically insert prefetch instructions in this kernel. As such, we turn to PGI's prefetching directives [7]. The directive

```
c$mem prefetch <var1>[,<var2>[,...]]
```

will cause the PGI compilers to emit prefetch instructions for the variables <`var`*n*>.

In our examples, the compilers emit `prefetchnta` by default. This is the desired instruction, for two reasons [2, 4]. This data is not reused, so eviction to L2 would possibly displace other, useful, data. Also, the arrays are much larger than 1 MB (see Table 2—the smallest of the arrays, namely `colidx` for the 256-process Class D benchmark, exceeds 10 MB by itself), so each array access will almost certainly miss in L2. If the `prefetchnta` misses in L2, then the prefetched data is not evicted into L2, avoiding cache pollution with unneeded data.

For this kernel, it is empirically best to prefetch three cache lines ahead for `a` (whose elements are 8 bytes), and two lines ahead for `colidx` (whose elements are 4 bytes). This could lead to the following code:

```
      do j=1,lastrow-firstrow+1
         sum = 0.d0
         do k=rowstr(j),rowstr(j+1)-1
c$mem prefetch a(k+24),colidx(k+32)
            sum = sum + a(k)*p(colidx(k))
         enddo
         w(j) = sum
      enddo
```

However, most of these prefetch instructions are unnecessary. We would prefer to emit only one prefetch instruction per cache line. Unrolling the loop to a depth of eight, as shown in Figure 2, will eliminate the superfluous prefetch instructions for `a`, but will still produce twice as many prefetch instructions as necessary for `colidx`. Unrolling further to another depth of two would eliminate the extra prefetches entirely, but this turns out to provide no speedup.

2.2 Reducing network contention

A huge volume of literature exists addressing data distribution for efficient matrix-vector products on various types of machines. Lewis and van de Geijn have published a number of papers examining such distributions for NPB CG in particular [5, 6]. Many of the methods they describe are promising candidates for implementation on XT systems, but would require significant redistribution of data after matrix generation.

Without having to redistribute the data, we can improve the communication characteristics by restructuring the work to avoid the global transposition. This transposition is the only communication step whose pattern cannot be mapped onto a three-dimensional torus without network contention.

Reference code | **Unrolled with prefetching**

```
do j=1,lastrow-firstrow+1                    do j=1,lastrow-firstrow+1
                                                i = rowstr(j)
                                                iresidue = mod( rowstr(j+1)-i, 8 )
   sum = 0.d0                                   sum = 0.d0
                                                do k=i,i+iresidue-1
                                                   sum = sum +  a(k)*p(colidx(k))
                                                enddo
   do k=rowstr(j),rowstr(j+1)-1                 do k=i+iresidue, rowstr(j+1)-8, 8
                                       c$mem prefetch a(k+24),colidx(k+32)
      sum = sum + a(k)*p(colidx(k))                sum = sum + a(k  )*p(colidx(k  ))
                                            &                + a(k+1)*p(colidx(k+1))
                                            &                + a(k+2)*p(colidx(k+2))
                                            &                + a(k+3)*p(colidx(k+3))
                                            &                + a(k+4)*p(colidx(k+4))
                                            &                + a(k+5)*p(colidx(k+5))
                                            &                + a(k+6)*p(colidx(k+6))
                                            &                + a(k+7)*p(colidx(k+7))
   enddo                                        enddo
   w(j) = sum                                   w(j) = sum
enddo                                        enddo
```

Figure 2: Unrolling the inner dot-product loop for efficient insertion of prefetch directives

In the case of a square distribution, the following algorithm exploits the data redundancy among the processes to replace the transposition with a broadcast.

1. Assign A_{ij} to process (i, j).
2. Assign a copy of x_j to each of processes (k, j), $1 \leq k \leq r$.
3. Compute $A_{ij}x_j$ on process (i, j).
4. Sum the resulting $A_{ij}x_j$ across process rows, leaving the result only on the diagonal process (i, i).
5. Broadcast the result from process (i, i) throughout its process column.
6. Process (i, j) now holds corresponding blocks of x and $y = Ax$.

Moreover, any dot products utilizing y can be computed locally prior to the broadcast, by performing the local reductions only on the diagonal processes, and performing the global reduction only among those same processes.

3 Results

The transformations described in Section 2 provide an approximate 35–40% gain in the benchmarks' self-reported Mop/s rates, shown in Table 6.

NPB Class	MPI processes	Reference code (Mop/s/process)	w/ prefetching… (Mop/s/process)	…and comm. changes (Mop/s/process)
B	2 (packed)	253.41	346.65	—
D	64	76.93	97.44	104.20
	256	105.58	134.22	149.55

Table 6: Performance results before and after code optimizations

Sparse matrix-vector multiplication		
	Reference code	Optimized code
L1 D-cache accesses	79067963239 ops	86717281996 ops
L1 D-cache misses that hit in L2	27189417496 fills	23683802004 fills
L1 D-cache misses that miss in L2	1361834048 fills	4908784328 fills
D-TLB misses	101407612 misses	101671410 misses
LD & ST per TLB miss	779.71 refs/miss	852.92 refs/miss
LD & ST per D1 miss	2.77 refs/miss	3.03 refs/miss
User time	210.228 secs	147.312 secs
Avg Time FPUs stalled	55.929 secs	55.352 secs
Avg Time LSs stalled	17.227 secs	9.232 secs
`prefetch` instructions dispatched	1 instr	10323838 instr
`prefetchnta` instructions dispatched	0 instr	4843730349 instr
hardware prefetches attempted	4608844727 ops	91644351 ops
hardware prefetches cancelled	55158840 ops	69451760 ops

Table 7: Hardware counter data for CG Class B, 2 concurrent processes, reported as per-process averages

Table 7 shows hardware counter data for runs using reference and optimized code. This data measures only the sparse matrix-vector products shown in Figure 2.

One might expect that nontemporal prefetching would cause a higher number of hits in L2, since `prefetchnta` is intended to reduce L2 pollution. The counter data, however, shows the opposite. In fact, hardware prefetching is hiding this effect. When the reference code executes, the hardware prefetcher is able to fetch the entries of `a` and `colidx` into L2. Thus, when the load-store unit (LSU) attempts to load entries of `a` and `colidx` for the FPUs, the loads would miss in L1 but hit in L2.

Contrariwise, when the optimized code executes, the explicit `prefetchnta` instructions preempt the hardware prefetcher, pulling the data directly into L1 instead of L2. This means that the LSU will hit in L1 more often. The numbers do not show a markedly higher percentage of L1 hits because the `prefetchnta` instructions themselves miss in both levels of cache before retrieving data from memory.

4 Conclusions

NPB CG presents two popular, and widely-studied, challenges in HPC applications: sparse matrix-vector multiplication, and distributed linear algebra. The reference implementation presents only naïve (that is, easily-coded) implementations of both, allowing significant speedups with minor code modifications.

Sparse linear algebra is widely recognized as a game of random memory accesses. While the random accesses are the limiting bottleneck for these kernels, we have seen here that the read-only stride-1 accesses cannot be ignored. In particular, one must note when there is little or no data reuse. The Opteron's exclusive cache structure allows data to bypass L2 entirely, both to and from memory, thus avoiding cache pollution; however, the hardware is dependent on the software containing the correct prefetch instructions to accomplish this.

Distributed matrix and vector operations can be implemented with a wide range of distributions, each requiring specific communication patterns. While hypercube topologies are amenable to any number of layout strategies, meshes and tori are more vulnerable to network conflict, particularly in transpose operations. We have seen that a simple modification allows us to avoid the transpose used in the reference implementation. This gives a large decrease in communication time, without requiring redistribution of data among the processes.

References

[1] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Athlon™ 64 and AMD Opteron™ Processors Rev 3.06*, September 2005.

[2] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 10h Processors (Quad-Core)*, June 2007.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Report RNR-94-007, NASA Advanced Supercomputing Division, March 1994.

[4] John Levesque, Jeff Larkin, Martyn Foster, Joe Glenski, Gary Geissler, Stephen Whalen, Brian Waldecker, Jonathan Carter, David Skinner, Helen He, Harvey Wasserman, John Shalf, Hongzhang Shan, and Erich Strohmaier. Understanding and mitigating multicore performance issues on the AMD Opteron™ architecture. Technical Report LBNL-62500, Lawrence Berkeley National Laboratory, March 2007.

[5] John G. Lewis, David G. Payne, and Robert A. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. In *Proceedings of the Scalable High Performance Computing Conference 1994*, pages 542–550, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[6] John G. Lewis and Robert A. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Supercomputing '93: Proceedings of the 1993*

ACM/IEEE Conference on Supercomputing, pages 484–492, New York, NY, USA, 1993. ACM Press.

[7] STMicroelectronics, Inc. *PGI® User's Guide*, February 2007.