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

Stephen Whalen
Cray, Inc.

September 8, 2007

1 Description of SP

1.1 High-level description

Crudely described, SP seeks a steady-state solution to discretized Navier-Stokes equations in three dimensions, with Dirichlet boundary data. The Alternating Direction Implicit (ADI) Method decouples the discretized problem into a set of three uncoupled systems of linear equations, one for each of the nondimensionalized spatial coordinates ξ, η, and ζ. SP uses the diagonal form of the Beam and Warming approximate factorization algorithm [1, 5], giving the coefficient matrices a scalar pentadiagonal structure [2].

The numerical problem itself is derived in detail in Chapter 3 of the NPB 1.0 Specification [1], where SP appears as "Benchmark 2" at the end of the chapter. The discretization uses finite differencing, requiring correction with numerical dissipation terms.

SP Class D uses a $408 \times 408 \times 408$ grid, and advances the solution through 500 time steps.

1.2 Implementational details

The reference MPI implementation uses the Bruno-Cappello multi-partition scheme to distribute the work over the processes [3]. This scheme partitions the problem domain into many subdomains, referred to as "cells," and then assigns multiple cells to each MPI process. Specifically, if there are p processes, with p a perfect square, then the Bruno-Cappello method subdivides the domain into $p\sqrt{p}$ cells. Then, $\sqrt{p}$ of the cells are assigned to each process in such a way that every coordinate plane intersecting the grid intersects exactly one cell on each processor. This method automatically enforces efficient load balancing, and allows for overlap of communication with computation [4].

For SP Class D on 256 processes, the $408 \times 408 \times 408$ grid is thus subdivided into 4096 cells of average size $(25.5)^3$; that is, each cell is $n_\xi \times n_\eta \times n_\zeta$, where each of n_ξ, n_η, and n_ζ may be either 25 or 26. Each dimension must then be augmented by 1 in each direction to allow for data exchange.

The reference implementation describes the local grids using the arrays `cell_size(1:3,1:n)`, `start(1:3,1:n)`, and `end(1:3,1:n)`. The first index of each of these arrays corresponds to a spatial dimension, and the second index indicates a particular cell assigned to this process. For cells having a face on the global domain boundary, the corresponding `start(d,c)` or `end(d,c)` is set to 1. As such, the loops in the computational kernels may begin with `start(d,c)` and end with `cell_size(d,c)-end(d,c)-1`, thereby skipping those nodes having fixed boundary values.

1.3 Profiles

A portion of the function-level sampling profile is shown in Table 1(a). No single subroutine dominates the runtime, but we shall focus our efforts on the top subroutine, `compute_rhs`.

Table 1(b) shows the `compute_rhs` line-level profile from the same run. This profile shows that the finite-differencing loops that calculate the directional fluxes consume the majority of the run time.

Samp %	Cum. Samp %	Samp	Imb. Samp	Imb. Samp %	Function
100.0%	100.0%	21407046	--	--	Total
21.3%	21.3%	4562247	256.72	1.4%	compute_rhs_
13.9%	35.2%	2973804	630.58	5.2%	x_solve_
10.7%	45.9%	2299151	411.94	4.4%	y_solve_
10.4%	56.4%	2235427	271.86	3.0%	z_solve_
8.3%	64.7%	1773426	139.55	2.0%	lhsz_
8.0%	72.7%	1710870	366.91	5.2%	lhsy_
7.5%	80.1%	1601252	346.11	5.3%	lhsx_

(a) Function-level profile

Samp %	Samp	Imb. Samp	Imb. Samp %	Group Function Source Line
100.0%	21407046	--	--	Total
21.3%	4562247	--	--	compute_rhs_ rhs.f
23.0%	1048809	143.09	3.4%	line.315
16.1%	733160	151.09	5.0%	line.196
15.3%	698678	178.79	6.2%	line.28
15.0%	684080	135.81	4.9%	line.72
8.9%	405243	103.02	6.1%	line.275
8.7%	395992	112.16	6.8%	line.395
7.8%	356392	100.84	6.8%	line.156

(b) Line-level profile for `compute_rhs`

Table 1: CrayPat sampling profile for SP Class D, 256 processes

```
c---------------------------------------------------------------------
c loop over all cells owned by this node
c---------------------------------------------------------------------
       do    c = 1, ncells
--
c---------------------------------------------------------------------
c         compute xi-direction fluxes
c---------------------------------------------------------------------
          do    k = start(3,c), cell_size(3,c)-end(3,c)-1
             do    j = start(2,c), cell_size(2,c)-end(2,c)-1
                do    i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [update rhs(i,j,k,m,c), 1 ≤ m ≤ 5, with finite differences over i]
                end do
             end do
          end do
--
c---------------------------------------------------------------------
c         compute eta-direction fluxes
c---------------------------------------------------------------------
          do     k = start(3,c), cell_size(3,c)-end(3,c)-1
             do     j = start(2,c), cell_size(2,c)-end(2,c)-1
                do     i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [update rhs(i,j,k,m,c), 1 ≤ m ≤ 5, with finite differences over j]
                end do
             end do
          end do
--
c---------------------------------------------------------------------
c         compute zeta-direction fluxes
c---------------------------------------------------------------------
          do    k = start(3,c), cell_size(3,c)-end(3,c)-1
             do     j = start(2,c), cell_size(2,c)-end(2,c)-1
                do     i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [update rhs(i,j,k,m,c), 1 ≤ m ≤ 5, with finite differences over k]
                end do
             end do
          end do
--
       end do
```

Figure 1: Excerpt from `compute_rhs` reference source code

2 Optimizations

2.1 Flux calculations

Although the flux loops' utilization of a regular computational stencil hints at an opportunity for cache tiling, the small size of the cells frustrates any such attempt. To promote cache reuse, we

```
c---------------------------------------------------------------------
c         compute xi-, eta-, and zeta-direction fluxes
c---------------------------------------------------------------------
          do    k = start(3,c), cell_size(3,c)-end(3,c)-1
             do    j = start(2,c), cell_size(2,c)-end(2,c)-1

                do    i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [fully update rhs(i,j,k,1,c) with finite differences over i, j, and k]
                enddo

                do    i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [fully update rhs(i,j,k,2,c) with finite differences over i, j, and k]
                enddo

                do    i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [fully update rhs(i,j,k,3,c) with finite differences over i, j, and k]
                enddo

                do    i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [fully update rhs(i,j,k,4,c) with finite differences over i, j, and k]
                enddo

                do    i = start(1,c), cell_size(1,c)-end(1,c)-1
                   [fully update rhs(i,j,k,5,c) with finite differences over i, j, and k]
                enddo

             enddo
          enddo
```

Figure 2: Loop structure for optimized flux computation

turn instead to straightforward outer loop fusion.

The inner loops of the reference code's three flux-calculating loop nests, outlined in Figure 1, update of each five separate components of the `rhs` array. This is a five-dimensional array, with the loop indices `i`, `j`, `k`, and `c` in the first, second, third, and fifth dimensions of `rhs`, respectively; each inner loop makes large-stride accesses through `rhs` by accessing `rhs(i,j,k,m,c)` for $1 \leq m \leq 5$. One might think of this as an unrolled inner loop over `m`, although each value of `m` would have a distinct loop body.

We can improve cache reuse, and reduce the number of such strided accesses, by fusing the ξ-, η-, and ζ-flux calculations into one loop nest over `i`, `j`, and `k`. This makes more efficient use of data, particularly the elements of the approximate solution `u`, while they are still in L1. If we adopt the view of an unrolled inner loop over `m`, then we can improve performance even more by interchanging the `i` and `m` loops. We accomplish this by fissioning the `i` loop so that all updates to `rhs(:,j,k,m,c)` for fixed `j`, `k`, `m`, `c` are applied before incrementing any of these variables. The resulting code is outlined in Figure 2. Incidentally, further promotion of the `m` loop toward the outer loops is detrimental to performance.

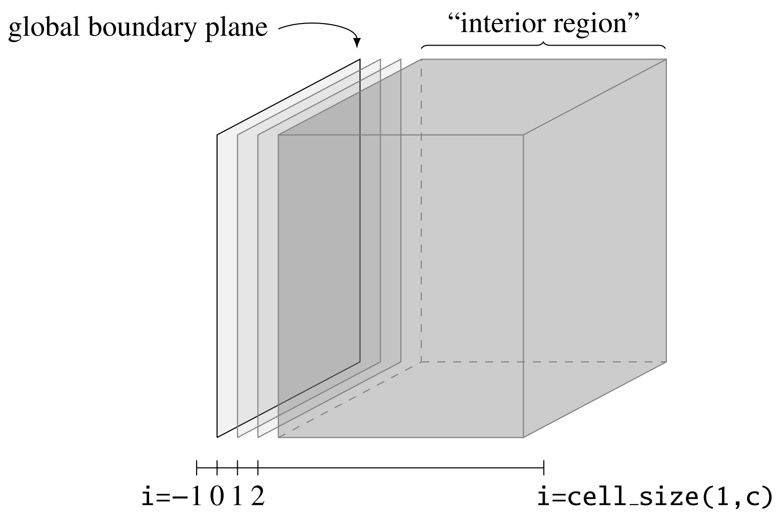


Figure 3: Special treatment of boundary conditions in computing numerical dissipation. Diagrammed here is the ξ-direction dissipation; the other directions simply require permutation of the indices.

2.2 Dissipation calculations

We can also attempt to fuse the dissipation loops, but the boundary conditions become troublesome. The numerical dissipation terms are computed independently in each coordinate direction. Away from the boundary, in the portion of the domain we shall call the "interior region," the numerical dissipation in each direction is a standard second-order finite difference for the second derivative. However, if one of the cell faces coincides with a domain boundary, then the next two parallel grid planes must use stencils that account for the boundary conditions. A cell having a boundary face is shown in Figure 3.

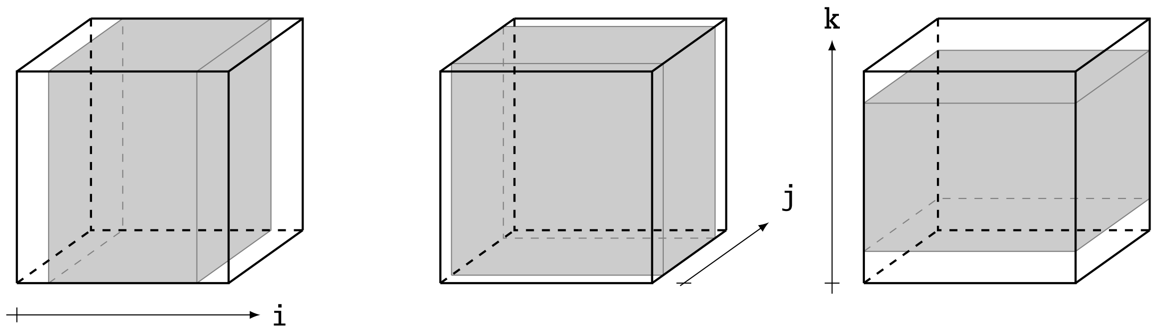
Potentially, in the single-process, single-cell case, a cell could have all of its faces on the domain boundary. Such a condition is shown in Figure 4(a). We can only fuse those portions of the loop nests that intersect, leaving six slabs that must be treated separately; see Figure 4(b).

When the cell size is sufficiently large, restructuring the dissipation loops in such a manner can give a performance gain. When the surface-to-volume ratio is sufficiently large, however, the extra loop overhead involved in treating the surface slabs separately erases any gains. We shall see that the 256-process case is still large enough to see some gain from this transformation.

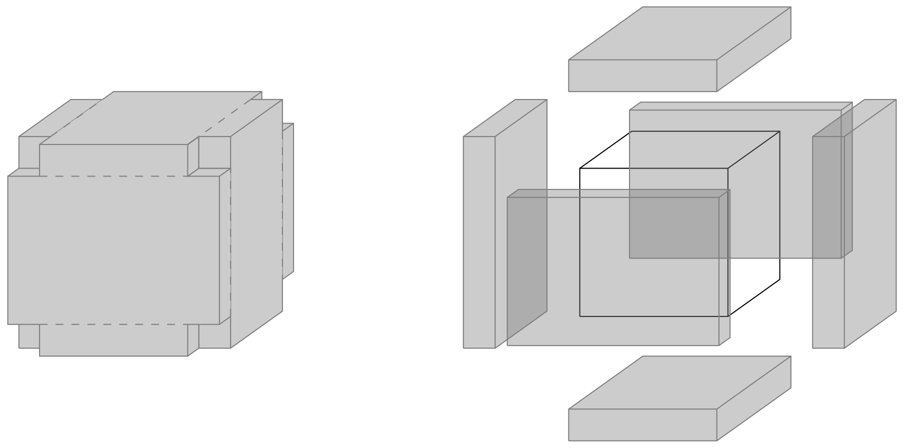
3 Results

The transformations described in Section 2 provide an approximate 7-10% gain in the benchmarks' self-reported Mop/s rates, shown in Table 2.

Table 3 shows hardware counter data for runs using reference and optimized revisions of the `compute_rhs` subroutine. By fusing loops, we are able to reduce cache activity altogether. The cache hit rates remain relatively constant, with an approximate 94% hit rate in L1. There are simply fewer accesses altogether: while the reference code visits the components of the `rhs` array at least once for each spatial dimension, the optimized loop nests visit each site only once for the



(a) Pictorial representation of the “interior regions” of a cell in each of the coordinate directions



(b) Union of the coordinate interior regions, together with exploded view showing the six slabs that cannot be fused into a single loop nest

Figure 4: Worst-case scenario in which a single cell intersects all six boundary faces of the global domain

MPI processes	**Reference code** (Mop/s/process)	**Fused flux loops** (Mop/s/process)	**Fissioned `i` loops** (Mop/s/process)	**New dissipation loops** (Mop/s/process)
64	368.75	386.38	399.09	406.42
256	364.06	375.47	387.53	391.01

Table 2: Performance results for NPB SP Class D, before and after code optimizations. Code changes are cumulative left to right.

`compute_rhs`		
	Reference code	Optimized code
L1 D-cache accesses	43275548856 ops	37948771376 ops
L1 D-cache misses that hit in L2	2369349249 fills	1805933638 fills
L1 D-cache misses that miss in L2	341857068 fills	241702538 fills
Evictions to L2	195421129158 bytes	146542456687 bytes
Evictions to memory	54361207123 bytes	36852743359 bytes
D-TLB misses	23964050 misses	17231166 misses
HW FP Ops	66515979393 ops	59450882504 ops
HW FP Ops / User time	869.829 M/sec	1087.613 M/sec
LD & ST per TLB miss	1805.85 refs/miss	2202.33 refs/miss
LD & ST per D1 miss	15.96 refs/miss	18.53 refs/miss
User time	76.470 secs	54.662 secs
Avg Time FPUs stalled	11.901 secs	7.112 secs
Avg Time LSs stalled	1.552 secs	1.486 secs

Table 3: Hardware counter data for SP Class D, 256 processes, reported as per-process averages

flux calculations, and once for the dissipation, except for points near the boundary.

4 Conclusions

Parallelization of the ADI Method has been studied extensively in the last thirty years, resulting in highly efficient domain decomposition strategies. The NPB SP benchmark demonstrates this efficiency through very small communication costs, waiting on communication for less than 5% of the total run time in the 256-process case.

Nevertheless, we have seen here that the serial portions of SP have left room for improvement. At least in the case of the `compute_rhs` subroutine, the reference implementation loops through the `rhs` array more often than necessary, creating superfluous cache activity. Fusing the multiple loops into larger loop nests reduces the required numbers of loads and stores, cutting the execution time of this subroutine by nearly 30%.

References

[1] 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.

[2] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. RNR Technical Report NAS-95-020, NASA Advanced Supercomputing Division, December 1995.

[3] John Bruno and Peter R. Cappello. Implementing the Beam and Warming method on the hypercube. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 1073–1087, New York, NY, USA, 1988. ACM Press.

[4] Rob F. Van der Wijngaart. Efficient implementation of a 3-dimensional ADI method on the iPSC/860. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 102–111, New York, NY, USA, 1993. ACM Press.

[5] R. F. Warming and R. M. Beam. On the construction and application of implicit factored schemes for conservation laws. In *Symposium on Computational Fluid Dynamics, SIAM-AMS Proceedings*, volume 11, 1978.