

Atomic integrals codes

How well do atomic integral codes perform?

Atomic integral codes in Dalton and NWChem hit between 10 and 150 Mflop/s, as measured both by large scale direct SCF calculations and small integral-only tests. The highest flop rate is observed for heavily contracted basis sets, while common Pople and Dunning basis sets hit approximately 50 Mflop/s. It is clear that the CPU spends very little time performing floating-point arithmetic.

Using the HPM counters, the instruction distribution of two-electron integral formation for a neon atom with the cc-pVTZ was produced.

instruction	rate (per cycle)
BGP_PU0_JPIPE_ADD_SUB	0.83
BGP_PU0_IPIPE_BRANCHES	0.37
BGP_PU0_IPIPE_ADD_SUB	0.30
total IPIPE/JPIPE	2.25
BGP_PU0_DCACHE_HIT	0.27
BGP_PU0_ICACHE_HIT	0.15
BGP_PU0_DATA_LOADS	0.15
BGP_PU0_DATA_STORES	0.05
BGP_PU0_FPU_OTHER_LOADS	0.05
BGP_PU0_FPU_OTHER_STORES	0.02
BGP_PU0_FPU_FMA_2	0.02
BGP_PU0_FPU_MULT_1	0.01

In short, atomic integral computation is primarily integer and logical operations followed by data movement, then instruction movement. Floating-point operations occur at a rate of less than 0.04 flop/cycle. While bookkeeping is necessary to compute atomic integrals, significant L1 traffic indicates register spilling due to fat inner loops and ~ 0.2 flop/load suggests unnecessary data access.

Schwarz screening saves flops but does it save time?

The following example is for a square array (R =edge) of neon atoms with the cc-pVDZ basis ($N_{AO} = 80$). The data structure required for Schwarz screening is N_{AO}^2 , which is larger than L3 cache for a non-trivial system. Because $(ij|ij)$ can be computed more easily than a general 2-electron integral quartet, Schwarz screening should be done only on the fly and only when necessary.

R	Gflop	Time	Mflop/s	Gflop	Time	Mflop/s	Flop ratio	Time ratio
1.0	5.3	142.2	37.2	5.3	142.9	37.3	99.4%	99.5%
2.0	4.2	112.2	37.3	4.6	121.6	37.8	90.8%	92.3%
3.0	1.8	47.4	37.2	2.7	65.4	40.9	65.9%	72.4%
4.0	0.7	19.5	37.2	1.4	26.8	50.6	53.4%	72.6%
8.0	0.4	11.2	37.2	0.9	12.9	68.3	47.3%	86.7%
16.0	0.3	8.9	37.2	0.8	10.5	75.9	41.6%	84.8%

Case study in tuning for BlueGene/P

Profiling results on multiple architectures revealed that a non-trivial amount of time in the triples evaluation of CCSD(T) is spent in hand-written loop code with complex memory access patterns.

	old	new		
Measure	TRPDRV	DGEMM	TENGY	TENGY
Gcycles	1.97	1.15	0.822	0.428
Gflop	3.12	2.94	0.173	0.166
Mflop/s	1345	1345	179.9	329.1

Extensive tuning lead to an assembly-intrinsic implementation which required approximately half the cycles for a 25% speed-up in the entire computation (not including communication). While this operation is far from peak floating-point performance due to the fixed ratio of flops per byte, the new kernel requires only 12.5% more cycles than the estimated theoretical minimum.

The second step, which is critical for BGP given the need for thread-assisted computation and the limited memory-per-node, was to multithread TENGY. The performance improvements relative to the optimal single-threaded version are 1.83 and 3.22 for two and four threads, respectively. The improvement of the optimal four-thread version relative to the original code is a near-perfect 3.88.

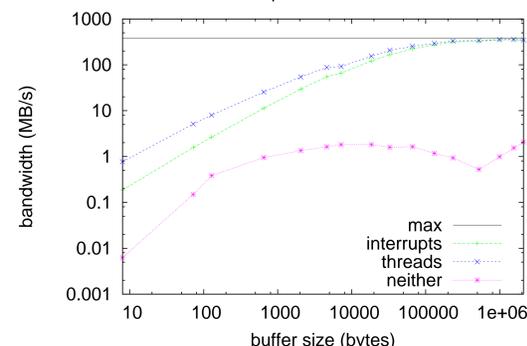
ARMCI Performance on BlueGene/P

Remote accumulate of non-contiguous memory is critical to quantum chemistry codes. Because DCMF does not support non-contiguous operations or accumulate, passive remote progress in ARMCI did not occur in the original implementation (See M. K. Krishnan, J. Nieplocha, M. Blocksome and B. Smith, "Evaluation of Remote Memory Access and Global Arrays Programming Model on the Blue Gene/P Supercomputer." In *Parallel Programming Models and Systems Software for High-End Computing*, 2008).

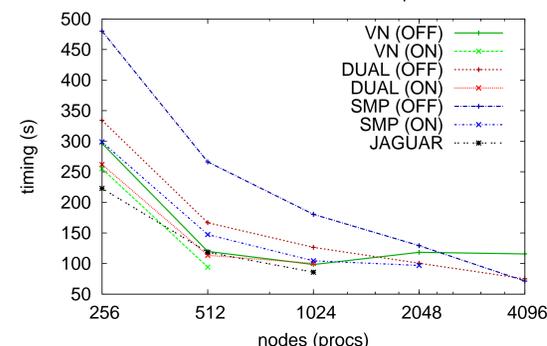
The first solution to the progress issue was to use "interrupt mode" which makes incoming messages a priority. However, an interrupt wipes the registers and kills performance of BLAS operations. The second solution was to spawn a thread to continuously invoke `DCMF Messenger_advance()` which ensures one-sided progress in ARMCI. This provides 65% (25%) of peak BW for contiguous (non-contiguous) messages of only 900 doubles.

We are currently working on extending the DCMF implementation of ARMCI to use more robust helper threads for communication offloading and active-messages for packing non-contiguous buffers.

ARMCI 2D Acc performance on BlueGene/P



NWChem - role of interrupts



The practical manifestation of improved ARMCI performance in the NWChem CCSD(T) code is shown above for the water octamer. Using interrupts improves the iterative time for the CCSD equations by 14-44% and the (T) step (not shown) by approximately 50%. Integration of thread-assisted ARMCI and newly-developed OpenMP parallelism in the (T) driver are expected to produce performance rivaling Jaguar (XT4) for the same number of nodes and cores despite the large discrepancy in memory-per-core and power consumption.

Tensor operations

Coupled-cluster methods are built upon tensor contractions. Large parallel calculations in the spin-orbital formalism require numerous array permutations in order to utilize DGEMM for the floating-point intensive portion of the contraction. Because it is generally impossible to compete with vendor-optimized BLAS libraries, we focus on the permutation portion of the tensor contraction. The general algorithm for the 4312 permutation is

```
forall ji = 0 to di - 1 (i = 1, 2, 3, 4)
  B(1 + j2 + d2 * (j1 + d1 * (j3 + d3 * j4))) = c * A(1 + j4 + d4 * (j3 + d3 * (j2 + d2 * j1)))
end
```

In this case, the optimal algorithm 1324 corresponds to j_4 as the innermost loop. While the general trend is obvious, the optimal algorithm — which gets within 10% of the performance of DAXPY — achieves a delicate balance between register usage and cache access. Neither maximizing L1 hit rate (for a fixed total) or minimizing L3 fetches alone determines the best loop ordering.

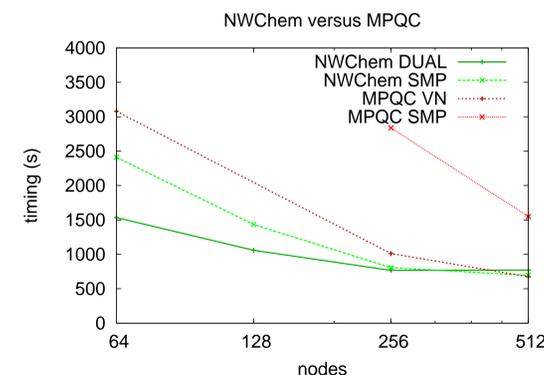
loop order	Gcycles	excess	L1 hits	L3 excess	permutation	speedup	best order	permutation	speedup	best order
2143	6.222	666.27%	36.62%	451.94%	1234	3.325	1234	3124	2.872	1432
4213	2.325	186.39%	0.42%	409.58%	1243	2.383	1234	4132	2.550	1432
1423	2.225	173.96%	0.65%	-29.92%	1342	2.288	1234	2341	2.646	2134
1432	1.881	131.65%	0.56%	-29.92%	2143	1.881	1234	4321	2.041	2134
1342	1.810	122.90%	0.47%	-34.54%	2314	2.958	1234	1423	2.617	2143
ORIGINAL	1.652	103.42%	2.92%	1.98%	3214	2.946	1234	3241	2.693	2314
3142	1.607	97.93%	0.77%	-30.65%	3142	2.345	1243	3421	2.065	2314
1243	1.151	41.69%	38.06%	2.03%	4123	2.682	1243	2431	2.750	2341
3241	0.963	18.61%	24.86%	28.85%	3412	1.819	1324	2134	2.972	2413
1234	0.894	10.10%	35.01%	5.62%	4312	1.857	1324	2413	2.576	2413
1324	0.893	10.01%	31.61%	5.62%	1324	2.989	1342	4213	2.208	2413
DAXPY	0.812	-	25.17%	-	1432	2.581	1342	4231	2.499	2413

The optimal algorithms for all possible 4-index permutations are given in the second table. The optimal permutation is highly architecture dependent. Of the 24 permutations, the optimal algorithm for PPC450 is the same as for Core2Duo in only 8 cases.

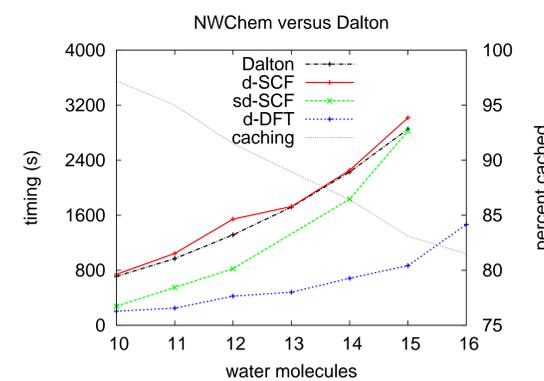
In addition to autotuning permutations, OpenMP parallelism has been added to the code. Threading efficiency is limited because these operations and memory-bound.

Comparative performance of SCF/DFT codes

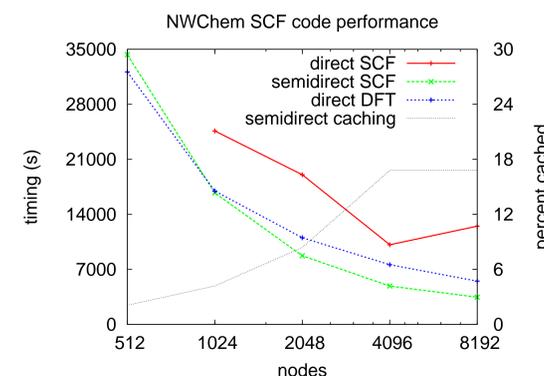
The first test was rubrene ($C_{42}H_{28}$) with the B3LYP/6-31+G* method. NWChem runs had thread support in ARMCI. MPQC used only one thread in SMP mode.



The second test was $(H_2O)_N$ with the SCF/aug-cc-pVDZ method.



Test is $(H_2O)_{64}$ with the SCF/aug-cc-pVDZ method (2624 b.f.).



Conclusions

1. *Big science requires different algorithms than small science.*

Example: Most quantum chemistry codes screen integral computation and distribute parallel tasks at the shell-quartet level. While this is effective at reducing flops, it adds branching and additional memory-access to inner-loops. The flops which are eliminated may require only a few thousand cycles, which may be less than the cost required to eliminate them.

Solution: Petascale chemistry calculations have hundreds of atoms and thousands of shell quartets. Screening (and parallel task distribution) should be done at the atomic scale since $O(10^8)$ atom-quartets exist. Only for extreme cases should branching and memory-access be traded for flops. Given the interatomic distances in large molecules, atom-level screening should be extremely effective at reducing flops.

2. *Code versatility should be implemented higher up in program flow.*

Example: the two-electron integral evaluation function — called $O(10^{10})$ (gigascale) to $O(10^{16})$ (petascale) times in the course of a calculation — should not check for a valid state and then branch into $O(10)$ possible integral evaluators. In 90% of jobs, the call path is invariant.

Solution: modern integral codes often are and should be generated automatically. Generated code should be integrated into the target program driver such that no branching is necessary. Program drivers need only be enumerated for valid and useful methods.

3. *Architectural variation requires drastic variation in algorithm.*

Example: while PPC450 is a quad-core processor with 8 MB L3 cache and 32 KB L1 dcache and icache per core, similar to commodity offerings, failure to address differences in register behavior and cycles-per-instruction may lead to significant performance degradation.

Solution: Legacy codes must reverse course and trade flops for registers, while new codes must enumerate inner-loop codes for at least different architecture types: embedded (e.g. PPC), commodity/server (e.g. Intel/AMD x86) and data-parallel/vector (GPUs). Scientific developers must properly abstract out inner-loops from program-flow to allow for independent tuning.

4. *Programming models must adapt to multithreading.*

Example: Within a single-thread, the one-sided progress assumption of ARMCI is not satisfied by the DCMF API on BlueGene/P. If we introduce communication threads to address this issue, lack of progress on a single thread in two-sided communication becomes less relevant. In addition, persistent all-to-all connectivity in one-sided is not scalable.

Solution: Explicitly multi-threaded codes which dynamically vary the number of communication and computation threads. The performance loss associated with communication threads decreases with the available threads-per-node, and few quantum chemistry codes approach theoretical peak anyways. Dynamic adaptivity in thread distribution ameliorates the performance loss associated with communication threads.

Description of Codes

NWChem was designed and written for massively parallel architectures starting in the 1990s by chemists and computer scientists at PNNL. NWChem uses a global-address-space, one-sided programming model in conjunction with dynamic load-balancing using GA/ARMCI.

Dalton evolved over many years from the codes developed in academic groups primarily located in Scandinavia. Master-worker parallelism was added to the SCF/DFT modules in a minimalist fashion using MPI (see D. Jonsson, K. Ruud, P. R. Taylor, *Comp. Phys. Comm.* **128**, 412-433 (2000) for a detailed description). Porting Dalton to the BlueGene/P platform required straightforward modifications from the options used for earlier IBM Power platforms and changes in I/O related to use of more than 1000 nodes and a shared file-system.

MPQC is developed at Sandia National Laboratory. In stark contrast to most quantum chemistry codes, MPQC is written exclusively in C++ and uses a hybrid programming model of MPI+threads for parallelism. The communication interfaces within MPQC are not yet optimized for BlueGene/P.

Acknowledgments

Manoj Krishnan (PNNL), Michael Blocksome (IBM) and Pavan Balaji (ANL) for help understanding and improving ARMCI over DCMF; Ed Valeev (VT) and Curt Janssen (SNL) for help porting MPQC to BlueGene/P; Graham Fletcher (ANL) and Nick Romero (ANL) for helpful discussions; Ray Bair (ANL) for encouragement.