QCD with Dynamical Fermions on the Connection Machine

Clive F. Baillie, Ralph G. Brichner
Rajan Gupta and S. Lennart Johnsson

YALEU/DCS/TR-735
September 1989

To appear in the Proceedings of Supercomputing 1989

QCD WITH DYNAMICAL FERMIONS ON THE CONNECTION MACHINE

CLIVE F. BAILLIE
Concurrent Computation Project, California Institute of Technology
Pasadena, CA 91125, USA

RALPH G. BRICKNER AND RAJAN GUPTA
Los Alamos National Laboratory
Los Alamos, NM 87545 USA

LENNART JOHNSSON
Thinking Machines Corporation
Cambridge, MA 02142 USA

We have implemented Quantum Chromo-Dynamics (QCD) on the massively parallel Connection Machine in *Lisp. The code uses dynamical Wilson fermions and the Hybrid Monte Carlo Algorithm (HMCA) to update the lattice. We describe our program and give performance measurements for it. With no tuning or optimization, the code runs at approximately 500 to 1000 MFLOPS on a 64-K Connection Machine, model CM-2, depending on the VP ratio.

Keywords: Quantum Chromo-Dynamics, QCD, Connection Machine, performance.

1. Introduction

We are part of a collaboration developing software to perform large-scale Quantum Chromo-Dynamics (QCD) computations on the massively parallel Connection Machine. This machine offers the potential of peak execution rates in excess of CRAY 2 or Y-MP systems currently available to us, so we have written a *Lisp version of a Fortran production code. The code is currently running in production mode on the CM-2, and we have obtained performance results for most of the *Lisp functions in the code. We have begun work on low-level arithmetic routines for high performance, which will be reported elsewhere. This paper first gives a brief description of the Connection Machine and an outline of QCD. Then we describe our dynamical fermion program in some detail. We end with timings and performance measurements obtained on a 64-K machine at Los Alamos National Laboratory.

2. The Connection Machine

The Connection Machine is a distributed-memory, single-instruction multple-data (SIMD) massively parallel processor comprising up to 65536 (64-K) processors.[1] Each processor consists of an arithmetic-logic unit (ALU), 8 kbytes of random-access memory (RAM), and a router interface to perform communications among the processors. There are 16 processors and a router per custom VLSI (very large scale integeration) chip, with the chips being interconnected as a 12-dimensional hypercube. Communications among processors within a chip work essentially like a cross-bar interconnect. RAM consists of standard 256-kbit chips. The processors deal with one bit at a time; therefore, the ALU can compute any two boolean functions as output from three inputs, and all data paths are 1-bit wide. In the current version of the Connection Machine (the CM-2), groups of 32 processors (two chips) share a 32-bit Weitek floating-point chip and a transposer chip that changes 32 bits stored bit-serially within 32 processors into thirty-two 32-bit words for the Weitek, and vice versa. The Connection Machine processors execute "nanoinstructions" broadcast to them by a sequencer that interfaces the Connection Machine to a front-end computer (currently a Symbolics 3600 LISP machine, a DEC VAX, or a Sun-4 workstation). The front-end computer reads and executes the user program, sending "macroinstructions" to the CM-2 sequencer through a special interface board.

The Connection Machine programming languages currently include the low-level "assembly" language Paris (Parallel instruction set), C* (a parallel version of C), CM Lisp, and *Lisp, with a parallel version of Fortran to appear in mid-1989. The highest performance has been obtained from *Lisp, so we have written our QCD programs in this language. The programming model of *Lisp[2] is conceptually very attractive: "pvars" (parallel variables) are new Lisp data types, which have an instantiation on each of the CM-2 processors. These "processors" are not necessarily the set of physical processors: virtual processor (VP) sets may be defined with multiple VPs per physical processor, limited only by the memory available. Parallel functions (indicated by the suffix "!!" on the function name) operate on the pvars on each processor in parallel. Conditional execution is achieved using a 1-bit "context flag," which selects certain processors and turns off the others during a given operation. Optimized global operations such as sums, maxima, and minima are provided with single function calls. The geometry of a particular problem is specified in N-dimensions very simply and at a high level (for example, a three-dimensional cube of grid points may be defined). Once this is done, communications among processors is accomplished via function calls.

3. Quantum Chromo-Dynamics

Quantum Chromo-Dynamics (QCD) is the gauge theory of the strong interaction that is responsible for binding quarks and gluons into hadrons (the constituents of nuclear matter). QCD can be investigated analytically using perturbation theory or numerically by computer simulation. However, perturbation theory for QCD is only valid at very high energies; hence, computer simulation is necessary for studying QCD at lower (experimentally attainable) energies. In order to do this, the four-dimensional space-time continuum is discretized onto a four-dimensional hypercubic periodic lattice of size $N = N_x \times N_x \times N_x \times N_t$, with the quarks living on the lattice sites and the gluons living on the links connecting the lattice sites. N_x is the spatial extent and N_t is the temporal extent of the lattice. Because including virtual quark anti-quark pairs in the simulation is very costly in terms of computer time, many simulations are done without these fermions, yielding "quenched" or "pure gauge" QCD. However, with the availability of very high performance machines such as the CM-2 and the CRAY-2 and Y-MP, we can include the virtual quarks and still run a simulation on a relatively large lattice. It is this "dynamical fermion" system that our code simulates. The gluons are represented by 3×3 complex $SU(3)$ matrices associated with each link in the lattice, while the Wilson representation of the fermions is used for the quarks. These consist of 3×4 complex matrices associated with each site on the lattice. During a QCD simulation, updating these link matrices consumes most of the computer time, with the fermion part comprising the vast majority. This is because all updating algorithms currently require inversion of a very large matrix at each step, and so must call a conjugate gradient (or alternative) routine inside the inner loop. There are several algorithms for doing this; our program uses the most efficient currently known - the Hybrid Monte Carlo Algorithm (HMCA), where a molecular dynamics (MD) algorithm is used to move the lattice through phase space, and step-size errors are corrected by a global Metropolis step after each MD trajectory.

Monte Carlo algorithms that simulate QCD generate an ensemble of configurations typical of statistical equilibrium for the system being studied. With this ensemble, one measures the physical observables of interest on each configuration and averages over the configurations. The error decreases as $\frac{1}{\sqrt{N}}$ for N configurations. The goal is thus to generate a series of configurations in the most efficient manner possible, and we use the HMCA.[3] In HMCA, we begin with the QCD action for the continuous theory:

$$E = S_G + S_F,$$

with the pure-gauge and fermionic parts being given by

$$S_G = \beta \sum_P (1 - \frac{1}{3} ReTrU_P).$$

and

$$S_F = \bar{\psi} M \psi,$$

Here

$$U_P = U_{i,\mu} U_{i+\mu,\nu} U^\dagger_{i+\nu,\mu} U^\dagger_{i,\nu}$$

is a product of link matrices around an elementary square or plaquette on the lattice (Figure 1), and

$$\begin{aligned} M[U]_{ij} &= \delta_{ij} + \kappa \sum_\mu [\ (\gamma_\mu - r) U_{i,\mu} \delta_{i,j-\mu} \\ &- (\gamma_\mu + r) U^\dagger_{i-\mu,\mu} \delta_{i,j+\mu}\ ] \end{aligned}$$

Figure 1: Calculation of elementary plaquette.

is the Dirac operator for Wilson fermions. In the following we set $r = 1$. In the discretized version of the theory, the action is written in terms of the psuedo-fermion fields ϕ:

$$S_F = \phi^\dagger (M^\dagger M)^{-1} \phi.$$

The HMCA consists of a series of molecular dynamics trajectories through some fictitious time, followed by a global Metropolis accept/reject. For the molecular dynamics evolution, the Hamiltonian is given by:

$$\begin{aligned} H &= \frac{\alpha}{2}\, Tr \sum P^2_{i,\mu} + \frac{\beta}{N} Tr \sum (1 - U_P) \\ &+ \phi^\dagger (M^\dagger M)^{-1} \phi\ . \end{aligned}$$

Here $U_{i,\mu}$ are the gauge link variables, $P_{i,\mu}$ are the momenta conjugate to them, and U_P is the 1×1 plaquette. The algorithm then consists of the following:

1. Start with a given set of U_μ.
2. Generate the P_μ as Gaussian random numbers ($Prob(x) = \exp -x^2$).
3. Generate r with probability $Prob\ (r) = \exp\ -\ r^2$.

Then, with

$$\phi = M^\dagger r, Prob(\phi) = \exp -\phi^\dagger (M^\dagger M)^{-1}\phi \quad ,$$

and the contribution of the fermions to the action is represented by Gaussian noise.

4. Leap-frog U_μ and P_μ through MD time, keeping ϕ constant.

5. Perform the global Metropolis step, to correct for the energy not being conserved because of the molecular dynamics approximation.

6. Go back to Step 2 and repeat.

Since M depends on the fields U, $\phi^\dagger(M^\dagger M)^{-1}\phi$ needs to be recalculated at every MD time-step, and thus the conjugate gradient (or alternative) routine needs to be called at every MD time-step. For some quark masses, better algorithms exist than conjugate gradient, and our code uses one of three built-in algorithms, each with a pre-conditioning step. These are conjugate gradient, and two versions of the minimal residual algorithm, with different pre-conditioners.

One feature of the gamma matrices enables us to reduce considerably the amount of arithmetic and communications required in the innermost loop. Noting that the γ_μ always appear as either $(1+\gamma_\mu)$ or $(1-\gamma_\mu)$ for the Wilson r value $r=1$, an inspection of the resulting matrices reveals that two rows of each can be reconstructed from the other two (or are zero, for γ_4). For example, for $(1+\gamma_\mu)$:,

$$\gamma_1 = \begin{bmatrix} 1 & . & . & i \\ . & 1 & i & . \\ . & -i & 1 & . \\ -i & . & . & 1 \end{bmatrix} \quad \gamma_2 = \begin{bmatrix} 1 & . & . & 1 \\ . & 1 & -1 & . \\ . & -1 & 1 & . \\ 1 & . & . & 1 \end{bmatrix}$$

$$\gamma_3 = \begin{bmatrix} 1 & . & i & . \\ . & 1 & . & -i \\ -i & . & 1 & . \\ . & i & . & 1 \end{bmatrix} \quad \gamma_4 = \begin{bmatrix} 2 & . & . & . \\ . & 2 & . & . \\ . & . & 0 & . \\ . & . & . & 0 \end{bmatrix}$$

and we see that, for example, the lower two rows of γ_1 can be reconstructed from the upper two by multiplying by $-i$.

Some of the observables one wants to measure in QCD are the string tension, the glueball mass spectrum, and the hadron mass spectrum. In order to do this the program calculates Wilson lines, and Wilson loops. Wilson loops are products of link matrices around closed rectangular paths on the lattice; Wilson lines are Wilson loops that wind completely around the lattice in a given direction. These quantities are measured on the lattice configurations and then averaged. Additional observables need to be measured However, these involve inversion of a fermion matrix, and may be done with existing analysis codes. Currently, we store our simulated lattices on a mass storage system at Los Alamos for later analysis.

4. The Dynamical Fermion Program

We have written a dynamical fermion QCD program in *Lisp for the CM-2. It employs a five-dimensional geometry: the four space-time dimensions, plus an additional dimension in which the coupling constant varies. Thus, we are able to run multiple copies of small lattices on a large machine, with different coupling constants and quark masses in each. Of course, communications among lattice sites are only done in the first four dimensions! There is a potential load balancing problem running multiple quark masses, since the inversion algorithms need to iterate different numbers of times for different quark masses. The geometry is generated by the Connection Machine operating system at a high level, so both setting up the geometry and performing the communications is done with very little effort on the user's part. All communications in the code are nearest-neighbor, and they are done only during the calculation of the gluon plaquettes, during calculation of M or $M^\dagger$ acting on psuedo-fermion fields, or during calculation of the Wilson loops and Wilson lines.

Basic operations in the above algorithm include a large variety of matrix arithmetic on 3×3, 3×4, and 3×2 complex matrices. A large library of such routines was written in *Lisp, including functions "su3ab," "add-mat," "m33," "sdot," "cdot," "saxpy34," and others. Functions that compress and expand full 3×4 matrices using the gamma matrix characteristics as described above are "proj-xp" and "expand-xp," plus variants for the $(1 - \gamma_\mu)$ matrices and for the y, z, and t directions. One level up, routines were constructed to calculate quantities such as $M^\dagger \phi$, $M\phi$, and $M^\dagger M \phi$ (e.g., "dslashsq-PP"). One higher level function "mdminv" calculates $\chi = (M^\dagger M)^{-1}\phi$, which calls the conjugate gradient or minimal residual solvers ("cg," "mr," or "mr2"). A level yet above evolves the system through molecular dynamics time. This calls the pure-gauge parts of the code (e.g., "gather-x," which calculates U_P) as well as the fermionic parts, and does the global Metropolis step. The main iteration loop also calls functions that calculate Wilson lines and Wilson loops; these are not part of the HMCA update process, but they provide some data for calculating observables. We also call a check-point function at this level, which writes out the lattice and other pertinent data either to a data vault or the front-end file system.

To summarize, the following functions comprise the bulk of the code:

- "swexp(swlog())" (software version of log and exponential functions),
- "su3ab" (multiply two $SU(3)$ matrices),
- "add-mat" (add two 3×3 matrices),
- "fetch-add" (get a 3×3 matrix from another processor and add to another matrix),
- "m33" (multiply two 3×3 matrices),
- "m33h" (multiply 3×3 matrix by adjoint of another),
- "adjoint-minus" (find negative of the adjoint of a 3×3 matrix),
- "phigen" (generate Gaussian random vector),
- "zero-mat34" (zero out a 3×4 matrix),
- "sdot" (dot product for real vectors),
- "cdot" (dot product for complex vectors),
- "add-mat34" (add two 3×4 matrices),
- "saxpy34" (real scalar times 3×4 matrix plus 3×4 matrix),
- "csaxpy34" (complex scalar times 3×4 matrix plus 3×4 matrix),
- "m34" (multiply 3×3 by 3×4 matrix),
- "m34h" (multiply 3×3 by adjoint of 3×4 matrix),
- "proj-xp" (project 3×4 matrix onto 3×2 matrix),
- "expand-xp" (expand 3×2 matrix into 3×4 matrix),
- "expand-set-xp" (expand 3×2 matrix into 3×4 matrix),
- "expand-add-xp" (expand 3×2 matrix into 3×4 matrix and add to matrix),
- "m32-v34" (3×3 matrix times 3×2 matrix into 3×4 matrix),
- "m32-v32" (3×3 matrix times 3×2 matrix into 3×2 matrix),
- "gather-x" (calculate plaquettes for x direction),
- "pgen" (generate Gaussian random vectors),
- "uup" (MD update of link matrices),
- "wilson-lines" (calculates Wilson lines),
- "wilson-loops" (calulates various Wilson loops),
- "wilson-loop11" (calculates the 1×1 Wilson loop),
- "tahpp" (traceless anti-hermitian part of MD momentum),
- "fpdot" (fermion contribution to MD time derivative of momentum)
- "calc-gpdot" (gluon contribution to MD time derivative of momentum),
- "cbarc" (calculate observables for diagnostics),
- "dslashsq-pp 0" (calculate positive part of $M^\dagger M \phi$),
- "dslashsq-pm 0" (calculate negative part of $M^\dagger M \phi$),
- "dslashsq-pp 1" (calculate positive part of $M\phi$),
- "dslashsq-pm 1" (calculate negative part of $M\phi$),
- "dslashsq-pm 2" (calculate positive part of $M_\dagger \phi$),
- "dslashsq-pm 2" (calculate negative part of $M_\dagger \phi$),
- "mr" (minimal residual solver),
- "mr2" (second minimal residual solver),
- "cg" (conjugate gradient solver).

With this outline of the dynamical fermion program one can understand the timings given in the next section.

5. Timings and Performance Measurements

We have timed essentially every function in the dynamical fermion QCD program, and worked out GFLOPS for some whose floating-point operation count we know. The timings are in Table 1, and the GFLOPS rates are given in Table 2. All the timed code was written in *Lisp, with no calls to Paris, except for "ranf!!" which calls a Paris fast random number generator. All the functions compiled (*compiled, in CM terminology). The compiler safety option was set to zero, so that no run-time checks were made on results. This produces the fastest executing code. Note that all arithmetic was done in 32-bit precision. The GFLOPS rates are calculated on the basis of the CM time. We can also calculate the performance using the front-end times but since almost all of the functions run at over 95% CM utilization this does not make much difference. Only two functions exhibited very low CM utilizations - "wilson-lines" and "wilson-loops" - these functions do a great deal of I/O to the front-end and therefore spend most of their time waiting for this to complete. One other function, "expand-add-xp," gave a slightly low CM utilization of 85%; we suspect that the compiler has not produced very efficient code for this and so shall re-structure the *Lisp to remedy it. We used the Los Alamos 64K CM-2 for these timings (that is, a 6.7 MHz 64K CM-2, with a dedicated Sun 4/260 front-end running a single user process). We ran a 16^4 lattice at one value of the coupling constant on a quarter of the machine (16K processors) in order to get a VP ratio of 4, and then extrapolated the performance to a full machine running with the same VP ratio (and hence a $16^3 \times 64$ lattice, for example). We choose VP ratio 4 since that is the largest with which we can run, given the memory requirements of our code. As the code spends most of its time in the "dslashsq" routines, we can safely say that it runs *sustained* at around $1 GFLOPS$. As expected, the highest performance - 1.9 GFLOPS - is achieved by the basic matrix multiply "su3ab", "m34" and "m32-v34."

6. Conclusions

We have translated a Fortran version of a dynamical fermion QCD code into pure *Lisp, except for a Paris call to a fast random number generator. This code runs at approximately 500 to 1000 MFLOPS on a 64-K CM-2 depending on the VP ratio. With the exception of inserting declarations to allow the code to *compile, no tuning or optimization was attempted. Because of the large ratio of computation to communications and the need for only nearest-neighbor communications on a four-dimensional lattice, this application runs well on the CM-2. We are currently working on low-level math routines for the code, written in a assembler-like language. We are currently running in production mode, and expect to publish physics results in late 1989.

7. Acknowledgments

Work was performed under the auspices of the United States Department of Energy. In the progress of this work we have made use of Connection Machines at the Center for Applied Parallel Processing (Boulder, Colorado), Argonne National Laboratory (Argonne, Illinois), the Northeast Parallel Architectures Center (Syracuse, New York), Boston University (Boston, Massachusetts), Thinking Machines Corporation (Cambridge, Massachusetts), and Los Alamos National Laboratory (Los Alamos, New Mexico). We thank these organizations for making this work possible.

References

1. W. Daniel Hillis, *The Connection Machine* (MIT Press, Cambridge, MA, 1985); "Connection Machine Model CM-2 Technical Summary," Thinking Machines Corporation TMC Technical Report HA87-4, Cambridge, MA, 1987.

2. "*Lisp Reference Manual" and "*Lisp Reference Supplement," Version 5.0, Thinking Machines Corporation, Cambridge, MA, 1988.

3. "QCD with Dynamical Fermions," R. Gupta, A. Patel, C. Baillie, G. Guralnik, G. Kilcup, and S. Sharpe, Los Alamos National Laboratory unclassified report LA-UR-88-998 and CERN-TH-5276/89 (1989).

Table 1. Timings for most of the functions in the code (with 4 VPs).

Function	Times Called	Front-End Times (sec)	CM-2 Time (sec)	CM Utilization (%)
swexp(swlog())	100	109.41	109.41	100
su3ab	1000	24.51	23.94	98
add-mat	10000	27.24	27.09	99
fetch-add	10000	57.85	57.47	99
m33	1000	27.82	27.36	98
m33h	1000	33.72	33.09	98
adjoint-minus	10000	22.41	22.31	100
phigen	10	548.51	547.40	100
zero-mat34	10000	16.34	16.19	99
sdot	1000	6.66	6.49	97
cdot	1000	18.29	17.87	98
add-mat34	10000	36.23	36.09	100
saxpy34	1000	7.03	6.90	98
csaxpy34	1000	13.63	13.36	98
m34	1000	37.02	36.29	98
m34h	1000	44.89	44.14	98
proj-xp	10000	18.47	18.31	99
expand-xp	10000	12.95	12.93	100
expand-set-xp	10000	25.39	25.26	100
expand-add-xp	10000	57.88	49.36	85
m32-v34	10000	185.57	181.53	98
m32-v32	10000	189.41	184.66	97
gather-x	100	46.88	46.04	98
pgen	10	24.27	24.26	100
uup	100	20.00	19.67	98
calc-gpdot	1000	40.71	39.55	97
wilson-lines	1	4.08	1.40	34
wilson-loops	1	12.31	3.24	26
wilson-loop11	100	48.93	47.77	98
tahpp	1000	43.49	41.42	95
fpdot	100	59.08	57.18	97
cbarc	1	53.16	51.58	97
dslashsq-pp 0	100	40.96	40.61	99
dslashsq-pm 0	100	41.05	40.60	99
dslashsq-pp 1	100	20.76	20.54	99
dslashsq-pm 1	100	21.06	20.52	97
dslashsq-pp 2	100	21.05	20.53	98
dslashsq-pm 2	100	20.68	20.53	99
mr	1	0.52	0.49	94
mr2	1	1.36	1.31	96
cg	1	1.31	1.30	99

Table 2. GFLOPS rates for most of the functions in the code on 64-K CM-2 (with 4VPs)

Function	Operation Count	FLOPS/processor	GFLOPS/64-K
su3ab	174	7268	1.905
add-mat	18	6645	1.742
m33	198	7237	1.897
m33h	198	5984	1.569
sdot	36	5547	1.454
cdot	96	5372	1.408
add-mat34	24	6650	1.743
saxpy	48	6957	1.824
csaxpy	96	7186	1.884
m34	264	7275	1.907
m34h	264	5981	1.568
proj-xp	12	6554	1.718
expand-xp	6	4640	1.216
expand-set-xp	6	2375	0.623
expand-add-xp	30	6078	1.593
fetch-add	18	3132	0.821
adjoint-minus	9	4034	1.058
m32-v34	132	7272	1.906
m32-v32	132	7148	1.874
gather-x	2178	4731	1.240
dslash-pp 0	1752	4314	1.131
dslash-pm 0	1752	4315	1.131
dslash-pp 1	876	4265	1.118
dslash-pm 1	876	4269	1.119
dslash-pp 2	876	4267	1.119
dslash-pm 2	876	4267	1.119