QCD Calculations on the J-Machine

Robert Rippel
Department of Nuclear Physics
Technical University of Vienna
Vienna, Austria

Richard A.Lethin
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA

April 3, 1995

1 Introduction

Simulations in the area of Quantum Chromodynamics (QCD) are very time intensive. For calculations providing reasonably precise results the underlying algorithms have to be strongly optimized to guarantee full applicability in terms of execution time.
One possibility to further improve the execution behavior of these programs is to adapt them to parallel machines. This task needs a large amount of specific (and machine dependent) knowledge[6, 9] concerning data distribution[7] and communication behavior[8, 12] as well as methods for cache optimizations and other empiric data which has to be considered when optimizing on a specific parallel computer. Parts of of this knowledge can be used by applying already existing fine tuned routines for special machines[1] or classes of parallel machines[13, 3].
Especially for QCD two important aspects have to be considered:

- Fast communication has to be guaranteed on the target machine. During QCD calculations a large number of messages has to be sent and received. Slow communication modules would result in dead-locks of processors waiting for their input data of further computations. The speedup achieved by parallelization would be rather decreased.

- Floating point operations: Most QCD applications operate on fields of floating point numbers. Thus most operations are of type floating point eg. the multiplication of two Matrices of $SU(N)$ requires N^3 multiplication and $N^3 - N^2$ addition operations. Therefore the given target machine should strongly support all floating point operations.

2 QCD Algorithms

Different types of algorithms are applied in QCD, eg.

- Langevin
- Molecular dynamic (MD)
- Monte-Carlo (MC) and
- Hybrid algorithms

In our example we are closer looking toward an MD iteration algorithm. This lattice-QCD-algorithm is operating on a fourdimensional discrete grid with N links in each dimension.

3 The Algorithm

In this section the equation of motion and the discretization are briefly described.
For the discretization the classical equation of motion

$$\dot{a}_n^{x,\mu} = \frac{\partial H(a_n^{x,\mu}, p_n^{x,\mu})}{\partial p_n^{x,\mu}} = p_n^{x,\mu}$$

$$\dot{p}_n^{x,\mu} = -\frac{\partial H(a_n^{x,\mu}, p_n^{x,\mu})}{\partial a_n^{x,\mu}} = -\frac{\partial S(a_n^{x,\mu})}{\partial a_n^{x,\mu}}$$

will be iterated by the leap-frog method

$$a_n^{x,\mu}(t+\delta t) = a_n^{x,\mu}(t) + \dot{a}_n^{x,\mu}(t)\,\delta t = a_n^{x,\mu}(t) + p_n^{x,\mu}(t)\,\delta t$$

$$p_n^{x,\mu}(t+\delta t) = p_n^{x,\mu}(t) + \dot{p}_n^{x,\mu}(t)\,\delta t = p_n^{x,\mu}(t) - \frac{\partial S(a_n^{x,\mu})}{\partial a_n^{x,\mu}}\,\delta t$$

with initial values

$$a_n^{x,\mu}(0)\ ,\ p_n^{x,\mu}(0)$$

For calculation of S, we define

$$U^{x,\mu} = e^{-i\sum_{j=1}^{8} a_j^{x,\mu} \frac{\lambda_j}{2}}$$

There is

$$S = \sum_{x,\mu} S^{x,\mu}$$

$$S^{x,\mu} = \beta \sum_{\square} 1 - \frac{1}{3} real(trace(\mathbf{U}_{\square})) = \beta \sum_{\nu \neq \mu} 1 - \frac{1}{3} real(trace(\mathbf{U}_{\nu}^{\mathbf{x},\mu}))$$

where $U_\nu^{x,\mu} \in SU(3)$ is a plaquette spawned from site x in the directions μ and ν respectively (Fig.1).

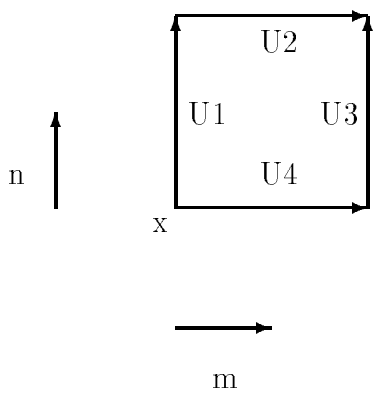

Figure 1: $U_n^{x,m} = \mathbf{U_1}\ \mathbf{U_2}\ \mathbf{U_3}^\dagger\ \mathbf{U_4}^\dagger$

Each matrix on a link represents a move from one site to the other one, in positive signed direction (eg. $\mathbf{U_1}$ and $\mathbf{U_2}$ in Fig.1).
Two moves are added by multiplication of the matrices. For a move in negative direction ($\mathbf{U_3}$ and $\mathbf{U_4}$ in Fig.1) the matrix must be taken c.c. and transposed due to the characteristic properties of $U \in SU(3)$.
This is verified by signing $\mathbf{U}$ as *crossed*: $\mathbf{U}^\dagger$.
The derivative of S is equal to

$$\frac{\partial S(a_n^{x,\mu})}{\partial a_n^{x,\mu}} = -\frac{\beta}{6}\frac{\partial}{\partial a_n^{x,\mu}}\sum_{\Box} real(trace(\mathbf{U}_\Box)) = ...$$

$$... = \frac{\beta}{6} trace(\mathbf{iT_n U^{x,\mu} V^{x,\mu}} - \mathbf{iV}^{\dagger\mathbf{x},\mu}\mathbf{U}^{\dagger\mathbf{x},\mu}\mathbf{T_n})$$

where T_n are the generators of $SU(3)$ and $V^{x,\mu}$ are the contributions of the six handles closing $U^{x,\mu}$ to a handle.

4 Virtual arrays

Virtual arrays are non-local arrays needed for calculation of the underlying algorithm. These arrays are distributed among all processors. Three different instances of *Virtual arrays* are required for discretization of the specific QCD algorithm:

- $U = \{U(x,y,z,t,\vec{e_i}) : 0 \le x < N, 0 \le y < N, 0 \le z < N, 0 \le t < N, 0 \le i < 4\}$
- $FO = \{FO(x,y,z,t,\vec{e_i}) : 0 \le x < N, 0 \le y < N, 0 \le z < N, 0 \le t < N, 0 \le i < 4\}$
- $PDTI = \{PDTI(x,y,z,t,\vec{e_i}) : 0 \le x < N, 0 \le y < N, 0 \le z < N, 0 \le t < N, 0 \le i < 4\}$

with $\vec{e}_i = \{\vec{e}_x, \vec{e}_y, \vec{e}_z, \vec{e}_t\}$
$U(x, y, z, t, \vec{e}_i)$ is a link between a point at location $\vec{x}$ and $\vec{x} + \vec{e}_i$. Each link is represented by a Matrix of $SU(3)$.
We are looking toward a fourdimensional periodic grid of size N^4. Due to the fact that each point is connected to eight neighbours, the number of links is $4N^4$ for every field.

5 Fundamentals for calculation

For any iteration, $FO'(t)$ is defined as

$$FO'(x, y, z, t, \vec{e}_i) = const \sum_{k \neq i} U_{\square}(x, y, z, t, \vec{e}_i, \vec{e}_k) + U_{\square}(x, y, z, t, \vec{e}_i, -\vec{e}_k) \tag{1}$$

with

$$U_{\square}(\vec{x}, \vec{e}_i, \vec{e}_j) = U(\vec{x}, \vec{e}_i)\, U(\vec{x} + \vec{e}_i, \vec{e}_j)\, U^{\dagger}(\vec{x}, +\vec{e}_j, \vec{e}_i)\, U^{\dagger}(\vec{x}, \vec{e}_j)$$

where eg.

$$U(\vec{x} + \vec{e}_y, \vec{e}_j) = U(x, y + 1, z, t, \vec{e}_j)$$

In the same sense $PDTI'(t)$ is defined as

$$PDTI'(x, y, z, t, \vec{e}_i) = PDTI(x, y, z, t, \vec{e}_i) + FO'(x, y, z, t, \vec{e}_i) \tag{2}$$

and $U'(t)$

$$U'(x, y, z, t, \vec{e}_i) = e^{-PDTI'(x,y,z,t,\vec{e}_i)} U(x, y, z, t, \vec{e}_i) \tag{3}$$

These fundamentals will be the base relations for the computation:

5.1 Initialization

At the beginning of all iterations U and $PDTI$ are initialized with certain values

$$U_0(x, y, z, t, \vec{e}_i)$$

and

$$PDTI_0(x, y, z, t, \vec{e}_i)$$

5.2 Iteration scheme

Applying formulas 1, 2 and 3 respectively, $FO'(t), PDTI'(t)$ and $U'(t)$ are calculated:

$$FO' = f(U_0)$$

$$PDTI' = f(PDTI_0, FO)$$

$$U' = f(U_0, FO)$$

Results of each iteration are stored in the field arrays and are the initial values for further iterations.

```
#define N 4
#define M 2
#define J 10

SU3 Ulocal[N^4/M^4*4],FOlocal[N^4/M^4*4],PDTIlocal[N^4/M^4*4];
int synch[N^4/M^4];

formula1Local(int length,int* arg)
  {
  <doing local calculation on each processor for FO'=f(U)>
  <including message passing>
  }

formula1()
  {
  int forSync;
  for(i=0;i<(N^4/M^4);i++)
    spawn(sizeof(int),&(synch[i]),i,formula1Local,sizeof(int),i);
  for(i=0;i<(N^4/M^4);i++)
    forsync=synch[i];
  }

<formula2 and formula3 analogous to formula1>

main()
  {
  <initialize U from file>
  <initialize PDTI from file>
  for(i=0;i<J;i++)
    {
    formula1();
    formula2();
    formula3();
    }
  <write results back to file>
  }
```

Figure 2: MD iteration

5.3 Final results

After the last iteration the resulting field

$$U_{final}$$

is stored.

6 Description of the J-machine

The J-machine[10] was developed at the *Massachusetts Institute of Technology (MIT)* by the Concurrent VLSI Architecture Project. It was designed to support fine-grain processes but also to provide on-chip associative memory[4]. Another import aspect concerning the J-machine is an available hardware support for process synchronization[5].
Other parallel machines support local communication with a significantly lower cost than communication. This very often results in a programming task of those machines in executing large pieces of code locally while in the same sense minimizing communication. Two major advantages of this machine are supposed to be:

- The J-machine is supported by strongly optimized communication modules which allow the programming system to perform far more communication than earlier machines.
- Communication and synchronization by hardware is supported by the programming system in order to allow the programmer easier development of concurrent programs without further complicating his task by additionally required manual synchronization or communication.

The J-machine provides a message-driven programming system based on GNU-C. This system exploits the special features available in the underlying architecture.

6.1 Architecture

The current version of the J-machine network at MIT consists of 1024 processors[11]. These processors are connected in a three dimensional grid. The maximum size of an allocated processor array is $8 \cdot 8 \cdot 16$ respectively in the dimensions x, y and z (Fig.3).

6.2 System Tools

In the programming system of the J-machine for message driven C a set of different tools is available. This set contains of a

- MDC Compiler
- MDC Linker
- Specific Libraries and Macros for MP

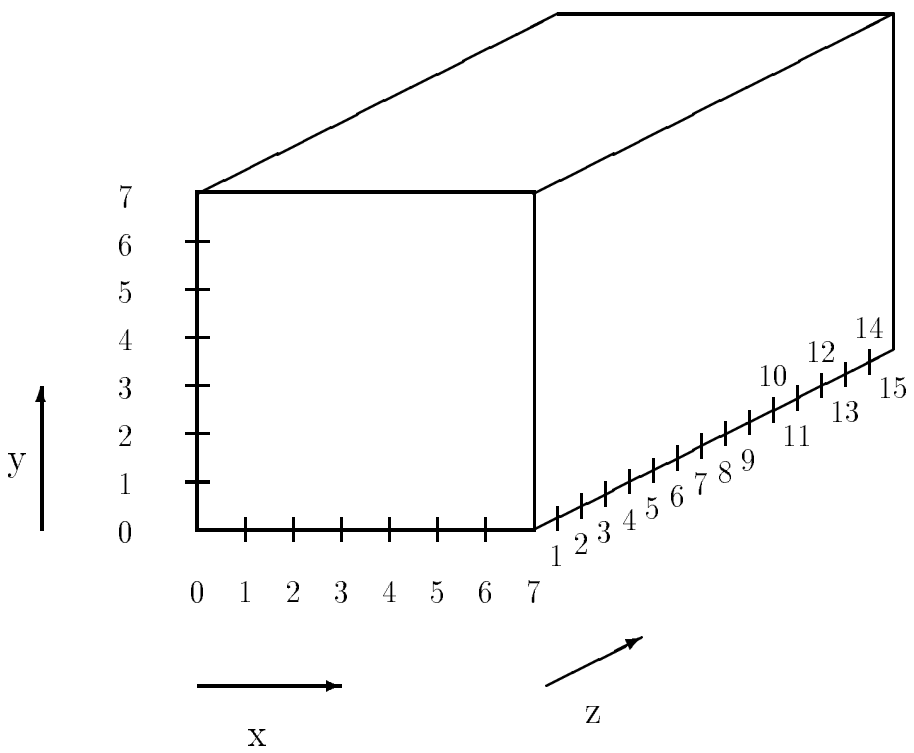

Figure 3: J-machine architecture

7 Implementation

The program for the explained algorithm is implemented in C (GCC) using extension macros for handling the message passing part on the J-machine. An idea of the control flow of the program can be taken from Fig.4:

- Initial values of the fields will be taken from the filesystem (a)
- Data distribution: All segments of arrays will be sent to owning processors (b)
- Local calculations including necessary communication will be performed (c)
- Resulting field configurations will be stored on the fileystem after having finished all iterations (d)

The whole program can be seen as pseudo code in Fig.2.

8 Data distribution

All data of *Virtual arrays* is distributed blockwise to the processor array (Fig.4b). Each processor owns the given points and all links connected to neighbours in positive directions. An example for the data distribution of all gridpoints for a fourdimensional array (with $N = 8$) distributed on 16 processors is shown below:

```
Data distribution for N=8
(Nr.of processors=16)

Node  0 is owner of point(0-3,0-3,0-3,0-3)
Node  1 is owner of point(4-7,0-3,0-3,0-3)
Node  2 is owner of point(0-3,4-7,0-3,0-3)
Node  3 is owner of point(4-7,4-7,0-3,0-3)
Node  4 is owner of point(0-3,0-3,4-7,0-3)
Node  5 is owner of point(4-7,0-3,4-7,0-3)
Node  6 is owner of point(0-3,4-7,4-7,0-3)
Node  7 is owner of point(4-7,4-7,4-7,0-3)
Node  8 is owner of point(0-3,0-3,0-3,4-7)
Node  9 is owner of point(4-7,0-3,0-3,4-7)
Node 10 is owner of point(0-3,4-7,0-3,4-7)
Node 11 is owner of point(4-7,4-7,0-3,4-7)
Node 12 is owner of point(0-3,0-3,4-7,4-7)
Node 13 is owner of point(4-7,0-3,4-7,4-7)
Node 14 is owner of point(0-3,4-7,4-7,4-7)
Node 15 is owner of point(4-7,4-7,4-7,4-7)
```

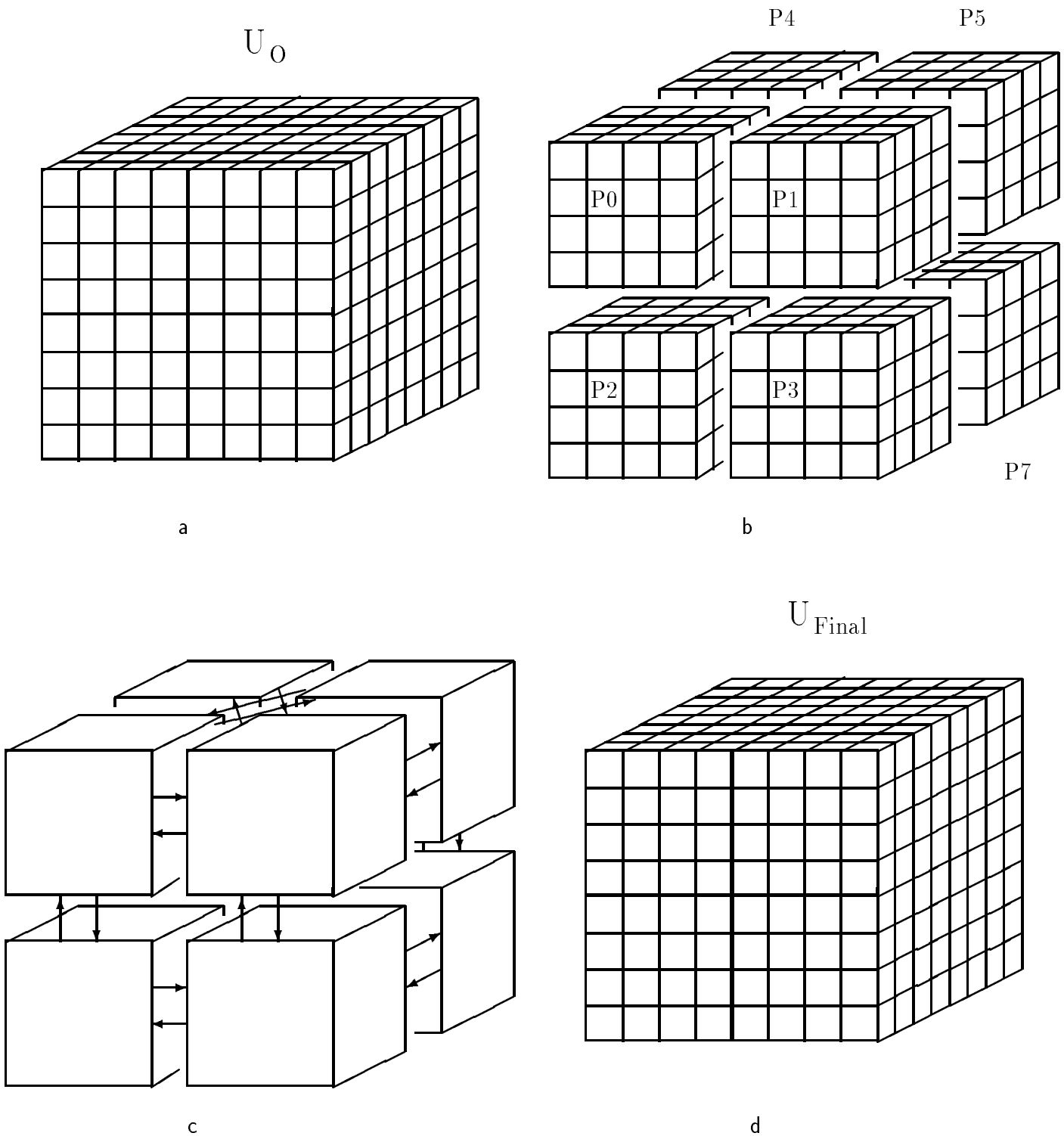

Figure 4: Data distribution and communication

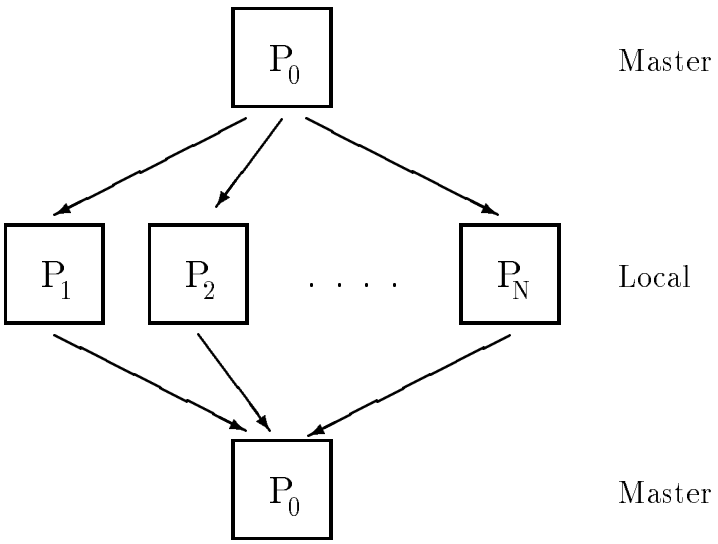

Figure 5: Master part and local part of a field-function

9 Task parallelism

Local calculations are proceeded according to the *owner computes rule*: Any assignment regarding to an element of an array is computed by the processor owning the element. Requested non-local elements of the same or another array are received by the processor via message passing and stored in a local buffer.
Any function operating on a field therefore consists of two parts:

- The **Master** part: This function spawns all local functions (explained below) to the associated processors and handles all necessary synchronizationof local calculation.
- The **Local** part: In this function all local computation on one processor is done. Non-local data will be accessed via message passing.

This idea is illustrated in Fig.5.
An example of such field operations is shown in Fig.2, where `formula1()` is associated with the master part and `formula1Local()` with the local part.

10 Synchronization by process suspension

On the J-machine synchronization can be done by the return values of spawned function calls[5].
Say we have a process p spawning a function f on processor n with return value ret: The return value ret will be tagged undefined until the spawned process f will be terminated on processor n and the return value is available in variable ret.
In Fig.6 an easy example for this suspension mechanism is given.

processP on processor p will be suspended and wait at this point for returnvalue of foo() on processor 10

```
processP()
{
int x,y,z,a,b,c,d;

x=5.3;
y=3.7;

aspawn(sizeof(int),&z,10,foo,sizeof(int),&x);

a=x+y;
b=x-y;
c=x+z;
d=x-z;
}



int foo(int len,int* arg)
{
return (2.0*arg);
}
```

Figure 6: Synchronization via process suspension

11 Communication

All communication is done by a special macro library[5] available for the J-machine. These macros make it easier to handle message passing.
In particular, software support by this macro library is available to

- figure out the size of allocated processors
- identify the operating processor
- spawn processes on other processors
- send and recieve data between spawned processes

11.1 J-machine macros

For our approach the following J-machine macros are used:

1. `computers()`

   This macro returns the number of allocated processors on the target machine

2. `computer()`

   Returns the identifying number of the current processor. This number is an integer in the range of 0 to `computers()`-1

3. `aspawn(sizeOfReturnValue, returnValuePointer, nrOfProcessor, spawnedProcess, sizeOfArgument, argumentPointer)`

   The function `spawnedProcess` will be spawned to processor `nrOfProcessor`, where the argument of length `sizeOfArgument` is pointed at by `argumentPointer`. The return value of lenth `sizeOfReturnValue` will be stored into `returnValuePointer`. The process spawned to another processor will be run asynchronously if the process has no return value (void). The process spawning another process to a processor will be suspended in case of a non-void function and will wait for the return value when this is first accessed after spawning the process.

11.2 Example program

The following example program in Fig.7 illustrates the use of the macros explained above. It will start on processor zero and send an element of the array `args` to each process being spawned. The returned number of processor will then be printed out.
The output of this program is (when run on 8 processors)

```
 Starting on processor zero.
Now on processor 1, argument is 3
back from processor 1
Now on processor 2, argument is 5
back from processor 2
Now on processor 3, argument is 7
back from processor 3
Now on processor 4, argument is 11
back from processor 4
Now on processor 5, argument is 13
back from processor 5
Now on processor 6, argument is 17
back from processor 6
Now on processor 7, argument is 19
back from processor 7
 Exit at processor zero
```

12 Measurement results

All measurements were performed depending on the following parameters:

- *Number of processors*: Size of allocated processor array
- *Number of gridpoints*: The number of used links is therefor $4N^4$ while N is the Number of gridpoints per dimension.
- *Number of iterations*. The actual number of iterations was 10 for all executions.

[1] estimated, could not be proceeded due to memory allocation problems

```
#include <stdio.h>
#include <string.h>
#include <spawn.h>

#define M 8   /* Nr of processors */

int localprocess(int len, int* argument)
  {
  printf("Now on processor %i, argument is %i \n",computer(),*argument);
  free_msg(argument);
                       /* deallocate memory for argument */
  return computer();
                       /* and return number of processor */
  }

main()
  {
  int returnvalue,m;
  int args[]={1,3,5,7,11,13,17,19};


  printf(" Starting on processor zero.\n");
  for (m=1;m<M;m++)
    {
    aspawn(sizeof(int),&returnvalue,m,localprocess,sizeof(int),&(args[m]));
                       /* spawning process "localprocess" to processor m */
    printf("back from processor %i\n",returnvalue);
    }
  printf(" Exit at processor zero \n");
  exit(1);
  }
```

Figure 7: An example program

Number of processors	Number of gridpoints	execution time [s]	$\frac{exec.time}{\frac{N^4}{M4}}$ [s]	speedup
1	16	166.3	10.3	1
1	256	2565.8	10.1	1
1	4096	39587.1[1]	9.6	1
16	16	12.0	12.0	13.8
16	256	177.4	11.1	14.5
16	4096	2711.1	10.6	14.6
256	256	17.9	17.9	143.3
256	4096	212.0	13.3	186.7

Figure 8: Execution times on the J-machine

Number of gridpoints	execution time [s] Sparc1+	Speedup compared to J-machine (one processor)	execution time [s] Sparc IPX	Speedup compared to J-machine (one processor)
16	5.2	31.9	3.0	55.4
256	81.1	31.3	47.9	53.6
4096	1320.2	30.0	793.1	50.1

Figure 9: Execution times on Sparc

The results in Fig.8 show a (nearly) linear speedup. The communication time is not significant and thus does not shrink the speedup for larger fields.
In comparism to that a sequential version of the same program was executed on sequential machines (Fig.9).

12.1 Profiling the sequential version

The sequential version run on a Sparc1+ is (about) 31 times faster than the sequential version on the J-machine.
Profiling the program (with 256 gridpoints) on Sparc1+ gives the actual time spent at various subroutines (Fig.10).
More than 80 % of the whole execution time is spent for floating point operations, multiplication and trigonometric functions etc. (functions `mult, sincos, add` and `expon`).
Function `mcount` is of the profiling process, `element` does all message passing for handling access to non-local elements on the J-machine, on Sparc (obviously) only local variables are accessed. Considering this, the actual times spent for local floating point computation will be slightly different when extrapolating profiling (Fig.11).
Because the J-machine does not support floating point operations by hardware this causes extremely high losses in terms of execution time.
Due to the fact, that more than 85 % of the actual computation time is spent on floating point operations on Sparc1+, skipping them should leed to a speedup factor of 4-5, if a

```
Total execution time: 78.66 seconds

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 71.3      56.07    56.07   194560     0.29     0.29  _mult [5]
  5.2      60.16     4.09                             mcount (80)
  3.8      63.12     2.96                             _sincos [11]
  3.3      65.70     2.59   351232     0.01     0.01  _element [9]
  3.2      68.25     2.55    71680     0.04     0.04  _add [12]
  3.1      70.67     2.42    10240     0.24     0.24  _expon [13]
```

Figure 10: Profile of execution times for most expensive functions

Execution time for floating point operations	Execution time for other computation
85.8 %	14.2%

Figure 11: Execution times (excluding profiling)

similar behavior of the J-machine is being expected. Measurements and comparism to the computation including floating point operations are shown in Fig.12.

13 Conclusions

For QCD calculations in the sense they were performed explained in this paper, two major aspects have to be considered evaluating the ability of the J-machine as a useful target machine.

13.1 Linear Speedup

The resulting speedup on the J-machine is rather linear. Due to very fast communication delays because of network traffic are not significant. Using the J-machine at AI Lab of the *Massachusetts Institute of Technology (MIT)* where these calculations were performed gave us a speedup of 725 when calculating on 1024 processors.

Number of processors	Number of gridpoints	execution time [s]	speedup compared to floating point version
1	256	517.1	4.9

Figure 12: Comparism of execution times when skipping floating point operations

Number of processors	$speedup_{final}$ compared to Sparc1+	$speedup_{final}$ compared to Sparc IPX
256	5.9	3.5
512	11.8	6.9
1024	25.6	13.7

Figure 13: Final speedup for QCD code

13.2 Support for floating point operations

The J-machine does not support floating point operations by the hardware, all operations are done by software emulation. This results in a slowdown factor of about 31 for the J-machine compared to the Sparc 1+ and about 53 compared to a Sparc IPX.
For our calculations performed on the J-machine the real speedup will be

$$speedup_{final} = \frac{speedup_{Jmachine}}{speedup_{sparc}}$$

The results of final speedups are shown in Fig.13.
The final speedup compared to a Sparc IPX for 1024 processors is about 13.
Compared to the speedup of the J-machine when running on 1024 processors this does not seem to be tremendous. The slowdown because of non-existent floating point support by hardware results in a moderate effective speedup. Nevertheless QCD or similar computations which took hours to be performed on sequential computers like the given Sparc stations can now be run through more than 10 times faster. This leads to a more efficient use and more flexibility of these calculations and will support users in the future to calculate more precise results while needing rather less execution time.

References

[1] V.Cherkassky, R.Smith: *Efficient Mapping and Implementation of Matrix algorithms on a hypercube*
The Journal of Supercomputing, Vol 2, pp 7-27, 1988

[2] G.O. Cook: *ALPAL - A Program to Generate Physics Simulation Codes from Natural Descriptions*
Lawrence Livermore National Lab. UCRL 102076, November 1989.

[3] CLASSPACK: *Basic Math Library*
User's Guide, Release 1.2, Kuck & Associates, IL, USA, 1992

[4] W.J.Dally, A.Chien, J.A.Fiske, G.Fyler, W.Horwat, J.Keen, R.A.Lethin,M .Noakes, P.Nuth, D.S.Wills: *The Message Driven Processor: An Integrated Multicomputer Processing Element*
Proceedings: International Conference on Computer Design, Cambridge, MA, USA, Oct. 1992

[5] Daniel Masket: *A Message-Driven Programming System for Fine-Grain Multicomputers*
Master Thesis, California Institue of Technology, Pasadena, California, February 1994

[6] G.C.Fox, S.W.Otto, A.J.Hey: *Matrix algorithms on the hypercube I: Matrix Multiplication*
Parallel Computing, 1987, pp 17-31

[7] M.Gupta, P.Banerjee: *Automatic Data Partitioning on Distributed Memory Multiprocessors*
Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, October 1990

[8] S.L.Johnson: *Communication efficient Basic Linear Algebra Computations on hypercube architecture*
Technical Report YALE/CSD/RR-361, Yale University, 1985

[9] S.L.Johnson, T.Harris, K.Mathur: *Matrix Multiplication on the Connection Machine*
Proc. Supercomputing 89, Nov 13-17 1989, Reno Nevada, ACM Press, pp. 326-332

[10] M.D.Noakes, D.A.Wallach, W.J.Dally: *The J-Machine Multicomputer: An Architectural Evolution*
Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993

[11] P.R.Nuth, W.J.Dally: *The J-Machine Network*
Proceedings: International Conference on Computer Design, Cambridge, MA, USA, Oct. 1992

[12] K.Knobe, J.D.Lukas, G.L.Steele: *Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines*
Journal of Parallel and Distributed Computing 8, pp. 102-118, 1990

[13] Kuo-Niang Chiang (Ed.): *MSC/NASTRAN Kernels, Testers and Timers*
The MacNeal-Schwendler Corporation, Los Angeles, USA, 1991