Implementing LQCD applications using SciDAC and Chroma

D. Pleiter

DESY

27. November 2006, LAP06, Zeuthen

Agenda

Agenda

Disclaimer and Credits

Disclaimer: **Not the speciality of the cook!**

Credits: Talk about work done by others
Here: USQCD collaboration
`http://www.usqcd.org`

Introduction

Challenges for efficient implementations

- Complex algorithms (see last talks)
- Complex computer architectures (see next talks)

Nice to face challenges, but: **Time-to-solution may become large!**

Could community efforts help?

- Interfaces to hide algorithmic details
- Interfaces to hide machine details (maschine abstraction)

USQCD started to go down this road. **What did they achieve?**

Overview on SciDAC Software

SciDAC software modules (level 1)

- **Q**CD **M**essage **P**assing (QMP)
- **Q**CD **L**inear **A**lgebra (QLA)

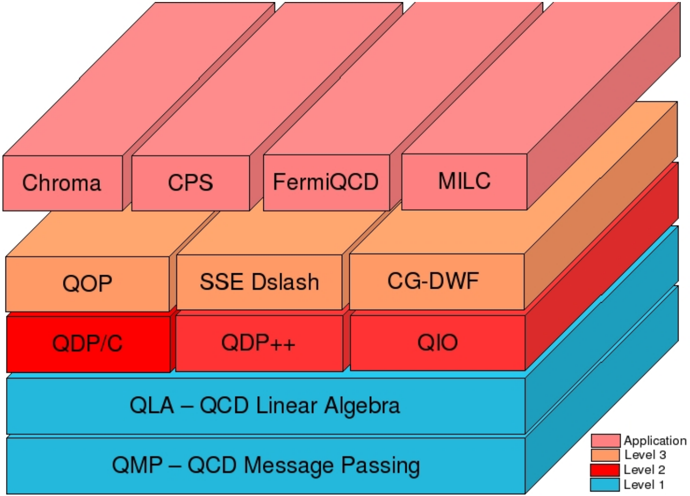
SciDAC software modules (level 2)

- **Q**CD **D**ata **P**arallel (QDP)
- **Q**CD **I**nput **O**utput (QIO)

Applications

- e.g. Chroma

Overview on SciDAC Software (2)



Agenda

MPI: Advantages and Drawbacks

- **Advantages:**
  - Mature and well understood
  - Widely supported formal standard (since 1992)
  - User interface: simple and efficient
  - Available on (almost) all platforms
- **Disadvantages:**
  - Many features are not required for LQCD applications
  - These features may nevertheless affect on performance
  - Possibly difficult to use all hardware features
    - Architectures may be suitable (BG/L) or less suitable (apeNEXT, QCDOC) for MPI

Design Goals

- Provide **lean, standard** communication layer **optimized for LQCD**
  - Simple communication patterns in LQCD
    - **Homogenious** communication patterns
    - **Nearest neighbour** communication
    - **Torus** topology
    - Collective operations: globus sum, broadcast
  - Any-to-any communication not required
- Suitable for very **efficient implementation** on a **variety of interconnects**
- **Performance requirements**:
  - Overlap of communication and computation
  - Multiple outstanding communication requests
  - Request for send in several directions in one call (QCDOC optimisation)

QMP Functionality

- Initialisation and finalisation, e.g.
  - `QMP_init_msg_passing`, `QMP_finalize_msg_passing`
- Discover machine topology, e.g.
  - `QMP_get_node_number`
  - `QMP_get_allocated_number_of_dimensions`
  - `QMP_get_allocated_dimensions`
- Logical machine
  - View of the allocated machine
  - `QMP_declare_logical_topology(const int *d, int n)`
  - `QMP_layout_grid` (dynamic layout)
- Communication
  - `QMP_declare_send_to`, ...
  - `QMP_declare_send_relative`, ...
  - `QMP_declare_multiple`
  - `QMP_start`, `QMP_is_complete`, `QMP_wait`
- Collective operations

Example (1)

```
int main(int argc, char* argv[])
{
  ...
  QMP_status_t status;
  QMP_thread_level_t req, prv;

  /* Initialise */
  req = QMP_THREAD_SINGLE;
  status = QMP_init_msg_passing(&argc, &argv, req, &prv);

  ...

  /* Cleanup */
  QMP_finalize_msg_passing();

  return 0;
}
```

Example (2)

```
#define NDIM 4

int main(int argc, char* argv[])
{
  int l[NDIM] = {4, 4, 4, 8};
  QMP_status_t status;

  ...

  status = QMP_layout_grid(l, NDIM);

  print_topology();
  ...
}
```

Example (3)

```
int main(int argc, char* argv[])
{
  int l[NDIM] = {4, 4, 4, 8};
  int g[NDIM] = {2, 1, 1, 1};
  QMP_status_t status;

  ...

  status = QMP_declare_logical_topology(g, NDIM);
  status = QMP_layout_grid(l, NDIM);

  print_topology();

  ...
}
```

Example (4)

```c
#define NDIM 4

void print_topology(void) {
  int nnodes, idim;

  if (QMP_get_node_number() != 0) return;

  nnodes = QMP_get_number_of_nodes();
  printf("#nodes   = %d\n", nnodes);
  {
    const int *s = QMP_get_subgrid_dimensions();
    printf("subgrid =");
    for (idim = 0; idim < NDIM; idim++)
      printf(" %d", s[idim]);
    printf("\n");
  }
}
```

QMP

Review

- Restriction to suitable subset of MPI
- Performance issues due to requirements wrt machine architecture

Agenda

Introduction

- Linear algebra operations on single node
- C binding
- Data objects:
  - Primitives:

`QLA_Real`	`R`	real
`QLA_Complex`	`C`	complex
`QLA_ColorMatrix`	`M`	$N_c \times N_c$ complex matrix
`QLA_DiracFermion`	`D`	four-spin, N_c complex matrix

...

  - Array of primitives

Example: Left multiplication by colour matrix

$$\phi^{a}_{\mu,i} = U^{a,b}_{\nu,i}\ \psi^{b}_{\mu,i}$$

```
#define N 1000
#define xmalloc(t, n) (t *) malloc(sizeof(t) * n)

int main() {
  QLA_ColorMatrix    *u;
  QLA_DiracFermion   *psi, *phi;

  u   = xmalloc(QLA_ColorMatrix, N);
  psi = xmalloc(QLA_DiracFermion, N);
  phi = xmalloc(QLA_DiracFermion, N);

  QLA_D_veq_M_times_D(phi, u, psi, N);
}
```

Example: Left multiplication by colour matrix (2)

- Variations: `QLA_T_eqop_M_times_T()`

`T` =	`V`	(1-spin, N_c colour spinor)
	`H`	(2-spin, N_c colour spinor)
	`D`	(4-spin, N_c colour spinor)
	`M`	($N_c \times N_c$ complex matrix)
	`P`	($4N_c \times 4N_c$ complex matrix)
`eqop` =	`[v]eq`	($=$)
	`[v]peq`	($+=$)
	`[v]meq`	($-=$)
	`[v]eqm`	($=-$)

- Adjoint colour matrix: `QLA_T_eqop_Ma_times_T()`

Example: Reductions

$$r = \sum |\phi|^2$$

```
#define N 1000
#define xmalloc(t, n) (t *) malloc(sizeof(t) * n)

int main() {
  QLA_Real             r;
  QLA_DiracFermion    *psi;

  psi = xmalloc(QLA_DiracFermion, N);

  QLA_r_veq_norm2_D(&r, psi, N);
}
```

Other Functions

- Functions with complex arguments (e.g. $\sqrt{z}$)
- Random numbers (uniform and Gaussian distribution)
- Spin/colour traces
- Multiplication with gamma matrices
- Ternary operations, e.g.
  - Addition with scalar multiplication $a \cdot b + c$
- ...

Review

- Quite exhaustive set of operations
- Various routines optimised for SSE, SSE2, PPC 440, PPC 440d

Agenda

Introduction

- Data-parallel programming environment
- Uses QMP for communication
- C++ binding (alternative: QDP/C)

Data Objects

- **Concept: Object types catagories**
  - Basic machine types (e.g. int, float, double)
  - Real world types (e.g. complex)
  - “Primitive types” (e.g. scalar, vector, matrix)
  - Grid type

Construct object types using C++ templates

```
typedef OLattice<
        PScalar<PColorMatrix< RComplex<float>,Nc>>>
        LatticeColorMatrix
```

- Grid type: `OLattice`
- Primitive types: `PScalar`, `PColorMatrix`
- Real world types: `Rcomplex`
- Basic machine types: `float`
- (Constants: `Nc`)

Data Objects (2)

- **Examples for available objects:**

`ColorMatrix`	$N_c \times N_c$ complex matrix
`LatticeColorMatrix`	array of colour matrices
`LatticeFermion`	N_s spin N_c colour spinor

Data Layout

- **Information relevant for controlling data layout:**
  - Number of dimensions
  - Lattice size
  - Number of threads
- **Functions to control this:**
  - `Layout::setLattSize(const multi1d<int>& size)`
  - `Layout::setSMPFlag({true|false})`
  - `Layout::setNumProc(n)`
- **Get layout dependent information:**
  - `Layout::sitesOnNode()`
  - `Layout::nodeNumber(x)`

Operations (1)

- Typical operation in QCD applications:

$$\psi_{\alpha i} = U_{ij}(\gamma_0)_{\alpha\beta}\phi_{\beta j}$$

- Implementation in QDP++:

```
LatticeFermion       psi
LatticeFermion       phi
LatticeColorMatrix   u

psi = u * Gamma(1) * phi
```

- Operation may involve Hermitian conjugate, e.g. $U_{ij}^{\dagger}$:

```
psi = adj(u) * Gamma(1) * phi
```

Operations (2)

- Shift operation: move data to neighbouring lattice site
  - May include communication
- Example: calculation of staples:

$$U_\nu(x+\mu) * U_\mu(x+nu)^\dagger * U_\nu(x)^\dagger$$

- Implementation in QDP++:

```
#define NDIM 4

int                            mu, nu;
multi1d<LatticeColorMatrix>    u(NDIM);
LatticeColorMatrix             s;

s = shift(u[nu], FORWARD, mu) *
    adj(shift(u[mu], FORWARD, nu)) *
    adj(u[nu]);
```

Example

```
int main(int argc, char *argv[]) {
  QDP_initialize(&argc, &argv);

  const int xl[] = {4,4,4,4};
  QDP::multi1d<int> l(QDP::Nd);
  l = xl;
  QDP::Layout::setLattSize(l);
  QDP::Layout::create();

  QDP::multi1d<LatticeColorMatrix> u(Nd);

  ...

  QDP::QDP_finalize();

  return 0;
}
```

Agenda

Introduction

- Input/Output Applications Programmer Interface
- Different file formats supported:
  - “Single file format”
    - I/O through single master node
  - “Partition file format”
    - I/O through a set of I/O nodes
    - Serve all nodes of corresponding partition
  - “Multi file format”
    - Write of temporary files to local disks
- QMP is used for data transport
- Support for structured file format: **LIME**

LIME

- LIME = **Lattice QCD Interchange Message Encapsulation**
- LIME file consists of one or more **messages**
- Each message consists of one or more **records**
- A record contains ASCII or binary data
- 144-byte header:
  - control word (8 byte)
  - record length (8 byte)
  - type string (128 byte)

Example

```
#include <stdio.h>
#include "lime.h"

main(int argc, char* argv[])
{
  FILE             *fp;
  LimeReader       *r;
  unsigned char    *buf;

  fp = fopen(argv[1], "r");
  r  = limeCreateReader(fp);

  limeReaderNextRecord(r);
  printf("First header = %s\n", limeReaderType(r));
}
```

LIME Interface Functionality

- Read operations (selection)
  - Create reader: `limeCreateReader`, `limeDestroyReader`
  - Header accessors: `limeReaderType`, `limeReaderBytes`
  - Pay-load read: `limeReaderReadData`
  - Skip to next record: `limeReaderNextRecord`
  - Seek within record: `limeReaderSeek`
- Write operations (selection)
  - Create writer: `limeCreateWriter`, `limeDestroyWriter`
  - Write operation: `limeWriteRecordHeader`, `limeWriteRecordData`, `limeWriterCloseRecord`
- Header creation

Review

- Efficient read operations possible
- Support for >2 GBytes file size
- Standard adopted by ILDG

Agenda

Introduction

- Application level software package
- Written in C++
- Dependencies: QDP++, QMP, libXML
- Optimised for various platforms (PC-Clusters, QCDOC, BGL)
- Applications
  - Pure gauge heatbath
  - HMC
  - Measurement

Chroma Modules

- **I/O module:**
  - Support for different gauge configuration formats (incl. ILDG)
- **Measurments module:**
  - Generation of colour sources
  - Propagator smearing
  - Calculation of correlation functions
- **Inverters:** CG, BiCGStab, SUMR, MINRES, ...
- **Eigensystems**
- **Various fermion and gauge actions**

Step 1: Generate SU(3) Configuration

- Create driver/parameter XML file, say `su3-in.xml`
- Execute application:

```
purgaug -i su3-in.xml -o su3-out.xml
```

XML Driver File (1)

Start with cold configuration:

```
<?xml version="1.0"?>
<purgaug>
  <Cfg>
    <cfg_type>UNIT</cfg_type>
    <cfg_file></cfg_file>
  </Cfg>
  ...
</purgaug>
```

XML Driver File (2)

Define update parameters:

```
<MCControl>
  <RNG> . . . </RNG>

  <StartUpdateNum>0</StartUpdateNum>
  <NWarmUpUpdates>2</NWarmUpUpdates>
  <NProductionUpdates>1000</NProductionUpdates>
  <NUpdatesThisRun>10</NUpdatesThisRun>
  <SaveInterval>10</SaveInterval>
  <SavePrefix>result</SavePrefix>
   . . .
</MCControl>
```

XML Driver File (3)

Define physical parameters

```
<HBItr>
  <GaugeAction>
    <Name>WILSON_GAUGEACT</Name>
    <beta>6.0</beta>
    <GaugeState>
      <Name>SIMPLE_GAUGE_STATE</Name>
      <GaugeBC>
        <Name>PERIODIC_GAUGEBC</Name>
      </GaugeBC>
    </GaugeState>
  </GaugeAction>
  ...
  <nrow>4 4 4 8</nrow>
</HBItr>
```

XML Driver File (4)

Define mearuments

```
<InlineMeasurements>
  <elem>
    <Name>POLYAKOV_LOOP</Name>
    <Frequency>1</Frequency>
    <Param>
      <version>2</version>
    </Param>
    <NamedObject>
      <gauge_id>default_gauge_field</gauge_id>
    </NamedObject>
  </elem>
</InlineMeasurements>
```

Step 2: Measurement of Hadron Observables

- Typical measurement application tasks:
  - Read a gauge configuration
  - Generate a colour source
  - Compute quark propogator (“invert” fermion matrix)
  - Calculate correlation functions
- Similar procedure as before
  - Create driver/paramter file, say `hadron-in.xml`
  - Execute application:
    `chroma -i hadron-in.xml -o hadron-out.xml`

XML Driver File (1)

Read configuration:

```
<?xml version="1.0"?>
<chroma>
  <Cfg>
    <cfg_type>SCIDAC</cfg_type>
    <cfg_file>su3.lime1</cfg_file>
  </Cfg>
  ...
</chroma>
```

XML Driver File (2)

Define tasks:

```
<Param>
  <InlineMeasurements>
    <elem>
      <Name> . . . </Name>
      <Frequency> . . . </Frequency>
      <Param>
        . . .
      </Param>
      <NamedObject>
        <gauge_id>default_gauge_field</gauge_id>
        <source_id>src</source_id>
      </NamedObject>
      . . .
```

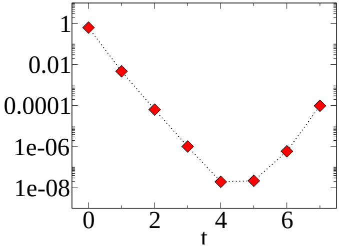
XML Driver File (3)

Calculate propagator:

```
<elem>
  <Name>PROPAGATOR</Name> ...
  <Param> ...
    <FermionAction>
      <FermAct>WILSON</FermAct>
      <Kappa>0.11</Kappa> ...
    </FermionAction>
    <InvertParam>...</InvertParam>
  </Param>
  <NamedObject>
    <gauge_id>default_gauge_field</gauge_id>
    <source_id>src</source_id>
    <prop_id>prop</prop_id>
  </NamedObject>
</elem>
```

Results

Nucleon correlator:



Review

- Some very nice concepts
- Toolbox for quick experiments
- Complex software, difficult to adapt to own needs
- Compiler dependencies
- Lack of documentation
- Tutorial: `http://www.ph.ed.ac.uk/~bj/HackLatt06`

Agenda

Conclusions

- **Advantages of community codes:**
  - Avoid re-implementation of standard tasks
  - Share porting and optimisation efforts
    - Highly optimised assembly code
  - Common lower level routines and data formats
    - Simpler to share code and data
- **Disadvantages of community codes:**
  - A lot of effort required to setup such an infrastructure
  - Some efforts may be needed to obtain good performance
    - Human resources vs. machine resources
- **SciDAC example:**
  - Very useful software infrastructure → Basis for applications software of most US-collaborations
  - Successful efforts to keep software
    - portable
    - (relatively) easy to install
  - Application level software more difficult to re-use

References

- Software installed in `/afs/ifh.de/group/ape/lap06/scidac`
- Copy of sources in `/afs/ifh.de/group/ape/lap06/scidac/src`
- For documentation see
  - `http://www.usqcd.org/usqcd-software`

Hints for Building Software

- Check file `INSTALL`
- Execute script `configure`
- When lucky (typically true): execute `make`
- Most software modules include examples