Grid Python Toolkit (GPT)

Christoph Lehner
(Uni Regensburg & Brookhaven National Laboratory)

https://github.com/lehner/gpt

September 22, 2021 - RIKEN CCS Seminar

Grid Python Toolkit (GPT)

- A toolkit for lattice QCD and related theories as well as QIS (a parallel digital quantum computing simulator) and Machine Learning
- Python frontend, C++ backend
- Built on Grid data parallelism (MPI, OpenMP, SIMD, and SIMT)

https://github.com/lehner/gpt

Initial commit Feb. 2020, 47k lines of C++/Python, >1400 commits so far, 12 contributors

Guiding principles:

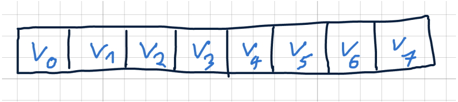
- Performance Portability
  common Grid-based framework for current and future (exascale) architectures

- Modularity / Composability
  build up from modular high-performance components, several layers of composability, “composition over parametrization”

The Grid data parallelism paradigm

https://github.com/paboyle/Grid

Start with a vector $v_x \in O$ with $x \in L$ and a d-dimensional Cartesian lattice L. Examples below have $d = 1$ and $L = \{0, \ldots, 7\}$.



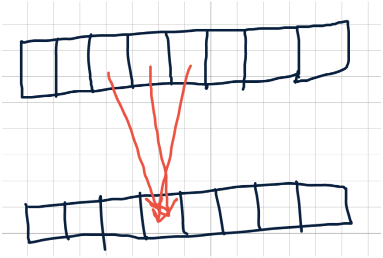
In lattice QCD, L makes up a space-time grid and v will be fermionic/bosonic fields.

For a dense state QC simulator of N qubits, $|L| = 2^N$, $O = \mathbb{C}$, and we first consider a canonical mapping of a state such as

$$\Psi = v_{\underbrace{000}_{=0}} |000\rangle + v_{\underbrace{001}_{=1}} |001\rangle + \ldots + v_{\underbrace{111}_{=7}} |111\rangle \,. \qquad (1)$$

High-performance building block: small stencil operators

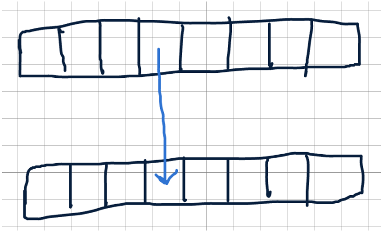
Common in lattice QCD: local operators with a small stencil (examples: Dirac matrix, Δ operator)



For such transformations, only knowledge of a few neighbors is needed.

High-performance building block: site-local operators

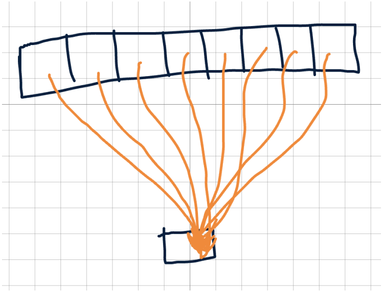
Examples: (bi-)linear combinations of vectors, R_ϕ gate



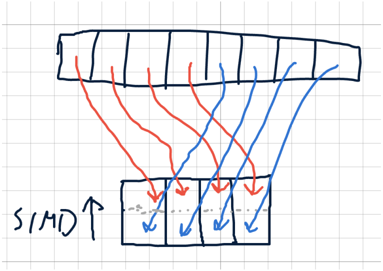
Definition: $R_\phi(v_0|0\rangle + v_1|1\rangle) = v_0|0\rangle + v_1 e^{i\phi}|1\rangle$

High-performance building block: reductions

Examples: inner product in lattice QCD, probability of measurement



For all these operations, the following data grouping preserves locality:



Such a group can be combined to a single SIMD word or mapped on a (fastest moving) thread index for coalesced memory access in SIMT architectures (Grid's SIMD/SIMT paradigm):

$$s_0 \equiv \begin{pmatrix} v_0 \\ v_4 \end{pmatrix}, \quad s_1 \equiv \begin{pmatrix} v_1 \\ v_5 \end{pmatrix}, \quad s_2 \equiv \begin{pmatrix} v_2 \\ v_6 \end{pmatrix}, \quad s_3 \equiv \begin{pmatrix} v_3 \\ v_7 \end{pmatrix} \tag{2}$$

Size of lattice of s reduces depending on SIMD word size.

Example: derivative on periodic lattice

The 8 operations

$$v_i' = v_{i+1 \mod 8} - v_i \tag{3}$$

with $i \in \{0, 1, \ldots, 7\}$ turn into 4 operations on SIMD words

$$s_j' = s_{j+1} - s_j \tag{4}$$

with $j \in \{0, 1, 2, 3\}$ and border permutation

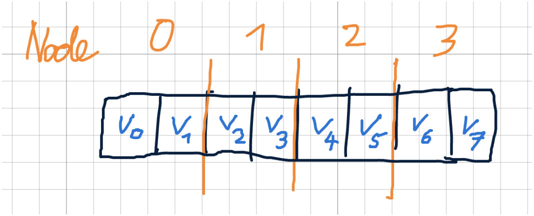
$$s_4 \equiv \begin{pmatrix} v_4 \\ v_0 \end{pmatrix} . \tag{5}$$

Check:

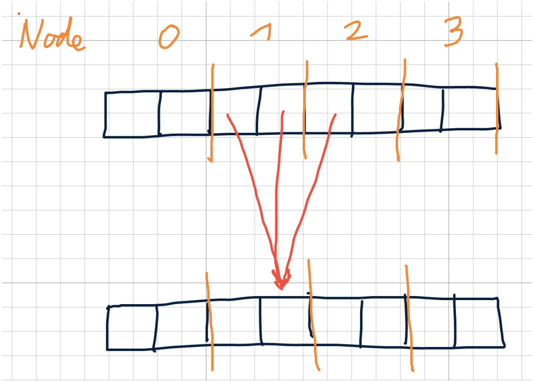
$$s_0 \equiv \begin{pmatrix} v_0 \\ v_4 \end{pmatrix} , \qquad s_1 \equiv \begin{pmatrix} v_1 \\ v_5 \end{pmatrix} , \qquad s_2 \equiv \begin{pmatrix} v_2 \\ v_6 \end{pmatrix} , \qquad s_3 \equiv \begin{pmatrix} v_3 \\ v_7 \end{pmatrix}$$

MPI parallelism

Here we allow for a d-dimensional Cartesian partition of the lattice L:



Challenge for Lattice QCD: small stencil operations

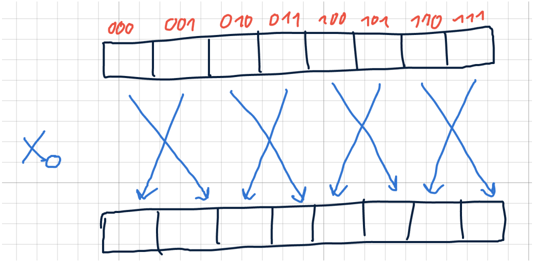


Only communication between neighboring nodes needed. Communication burden generally suppressed by surface to volume ratio.

Challenge for dense state QC simulator

Hadamard and CNOT gates can be mapped to site-local operations on original field and bit-flipped (X_i) fields, see later.

Non-locality therefore strongly depends on which X_i . . .

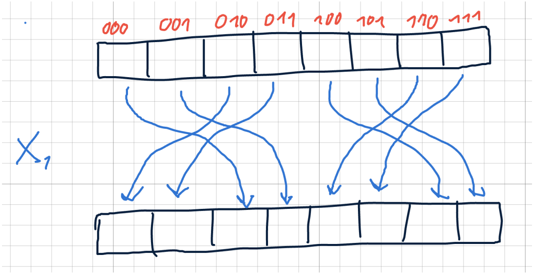


Operations may be maximally non-local!

Challenge for dense state QC simulator

Hadamard and CNOT gates can be mapped to site-local operations on original field and bit-flipped (X_i) fields, see later.

Non-locality therefore strongly depends on which X_i ...

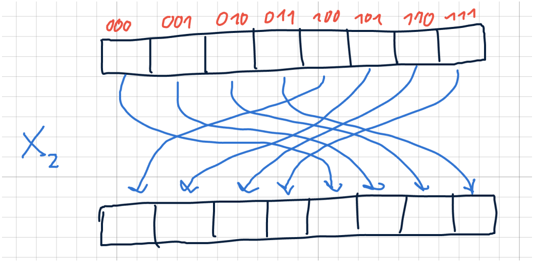


Operations may be maximally non-local!

Challenge for dense state QC simulator

Hadamard and CNOT gates can be mapped to site-local operations on original field and bit-flipped (X_i) fields, see later.

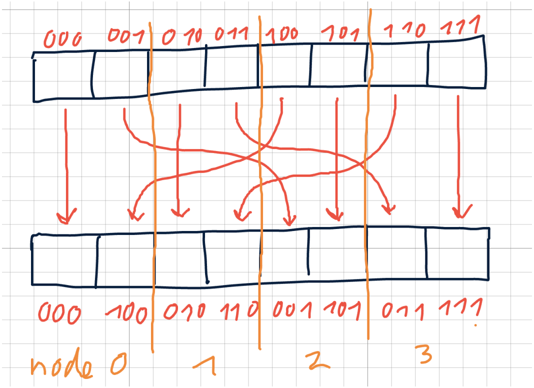
Non-locality therefore strongly depends on which X_i ...



Operations may be maximally non-local!

Dynamic qubit mapping

This problem can be mitigated (see also, e.g., JUQCS) by making qubit mapping dynamic. Example: order $b_2 b_1 b_0 \to b_0 b_1 b_2$



Subsequent X_2 gates now maximally local!

Smart grouping of gates in circuit and relatively infrequent memory layout changes can lead to significant speed up.

GPT - layout and dependencies

Python script / Jupyter notebook

gpt (Python)

- Defines data types and objects (group structures etc.)
- Expression engine (linear algebra)
- Algorithms (Solver, Eigensystem, ...)
- File formats
- Stencils / global data transfers
- QCD, QIS, ML subsystems

cgpt (Python library written in C++)

- Global data transfer system (gpt creates pattern, cgpt optimizes data movement plan)
- Virtual lattices (tensors built from multiple Grid tensors)
- Optimized blocking, linear algebra, and Dirac operators
- Vectorized ranlux-like pRNG (parallel seed through 3xSHA256)

Grid	Eigen	FFTW

The QCD module

Example: Load QCD gauge configuration and test unitarity

```
In [1]: import gpt as g

        U = g.load("ckpoint_lat.IEEE64BIG.1100")

        for mu in range(4):
            g.message("SU3 - Defect: ", g.norm2(U[mu] * g.adj(U[mu]) - g.identity(U[mu])))

GPT :       1.039211 s : SU3 - Defect:  3.345168726568745e-26
GPT :       1.094156 s : SU3 - Defect:  3.3476154954606903e-26
GPT :       1.146703 s : SU3 - Defect:  3.342180010368529e-26
GPT :       1.199097 s : SU3 - Defect:  3.3423193715873574e-26
```

Here: expression first parsed to a tree in Python (gpt), forwarded as abstract expression to C++ library (cgpt) for evaluation

Example: create a pion propagator on a random gauge field

```
# double-precision 8^4 grid
grid = g.grid([8,8,8,8], g.double)

# pRNG
rng = g.random("seed text")

# random gauge field
U = g.qcd.gauge.random(grid, rng)

# Mobius domain-wall fermion
fermion = g.qcd.fermion.mobius(U, mass=0.1, M5=1.8, b=1.0, c=0.0, Ls=24,
                               boundary_phases=[1,1,1,-1])

# Short-cuts
inv = g.algorithms.inverter
pc = g.qcd.fermion.preconditioner

# even-odd-preconditioned CG solver
slv_5d = inv.preconditioned(pc.eo2_ne(), inv.cg(eps = 1e-4, maxiter = 1000))

# Abstract fermion propagator using this solver
fermion_propagator = fermion.propagator(slv_5d)

# Create point source
src = g.mspincolor(U[0].grid)
g.create.point(src, [0, 0, 0, 0])

# Solve propagator on 12 spin-color components
prop = g( fermion_propagator * src )

# Pion correlator
g.message(g.slice(g.trace(prop * g.adj(prop)), 3))
```

Example: solvers are modular and can be mixed

General design principle: use modularity of python code instead of large number of parameters to configure solvers/algorithms;
Python can also be used in configuration files

```python
# Create an coarse-grid deflated, even-odd preconditioned CG inverter
# (eig is a previously loaded multi-grid eigensystem)
sloppy_light_inverter = g.algorithms.inverter.preconditioned(
    g.qcd.fermion.preconditioner.eo1_ne(parity=g.odd),
    g.algorithms.inverter.sequence(
        g.algorithms.inverter.coarse_deflate(
            eig[1],
            eig[0],
            eig[2],
            block=200,
        ),
        g.algorithms.inverter.split(
            g.algorithms.inverter.cg({"eps": 1e-8, "maxiter": 200}),
            mpi_split=[1,1,1,1],
        ),
    ),
)
```

Further example: Multi-Grid solver

```
def find_near_null_vectors(w, cgrid):
    slv = i.fgmres(eps=1e-3, maxiter=50, restartlen=25, checkres=False)(w)
    basis = g.orthonormalize(
        rng.cnormal([g.lattice(w.grid[0], w.otype[0]) for i in range(30)])
    )
    null = g.lattice(basis[0])
    null[:] = 0
    for b in basis:
        slv(b, null)
    g.qcd.fermion.coarse.split_chiral(basis)
    bm = g.block.map(cgrid, basis)
    bm.orthonormalize()
    bm.check_orthogonality()
    return basis

mg_setup_3lvl = i.multi_grid_setup(
    block_size=[[2, 2, 2, 2], [2, 1, 1, 1]], projector=find_near_null_vectors
)

wrapper_solver = i.fgmres(
    {"eps": 1e-1, "maxiter": 10, "restartlen": 5, "checkres": False}
)
smooth_solver = i.fgmres(
    {"eps": 1e-14, "maxiter": 8, "restartlen": 4, "checkres": False}
)
coarsest_solver = i.fgmres(
    {"eps": 5e-2, "maxiter": 50, "restartlen": 25, "checkres": False}
)

mg_3lvl_kcycle = i.sequence(
     i.coarse_grid(
         wrapper_solver.modified(
             prec=i.sequence(
                 i.coarse_grid(coarsest_solver, *mg_setup_3lvl[1]), smooth_solver
             )
         ),
         *mg_setup_3lvl[0],
     ),
     smooth_solver,
)
```

All algorithms implemented in Python – Example: Euler-Langevin stochastig DGL integrator

```python
class langevin_euler:
    @g.params_convention(epsilon=0.01)
    def __init__(self, rng, params):
        self.rng = rng
        self.eps = params["epsilon"]

    def __call__(self, fields, action):
        gr = action.gradient(fields, fields)
        for d, f in zip(gr, fields):
            f @= g.group.compose(
                -d * self.eps
                + self.rng.normal_element(g.lattice(d)) * (self.eps * 2.0) ** 0.5,
                f,
            )
```

Implemented algorithms:

- BiCGSTAB, CG, CAGCR, FGCR, FGMRES, MR solvers
- Multi-grid, split-grid, mixed-precision, and defect-correcting solver combinations
- Coarse and fine-grid deflation
- Arnoldi, implicitly restarted Lanczos, power iteration
- Chebyshev polynomials
- All-to-all vector generation
- SAP and even-odd preconditioners
- Gradient descent and non-linear CG optimizers
- Runge-Kutta integrators, Wilson flow
- Fourier acceleration
- Coulomb and Landau gauge fixing
- Domain-wall–overlap transformation and MADWF
- Symplectic integrators (leapfrog, OMF2, and OMF4)
- Markov: Metropolis, heatbath, Langevin, HMC in progress

Implemented fermion actions:

- Domain-wall fermions: Mobius and zMobius
- Wilson-clover fermions both isotropic and anisotropic (RHQ/Fermilab actions); Open boundary conditions available

Example: stout-smeared heavy-quark Mobius DWF

```
# load configuration
U = g.load(config)
grid = U[0].grid

# smeared gauge link
U_stout = U
for n in range(3):
    U_stout = g.qcd.gauge.smear.stout(U_stout, rho=0.1)

fermion_exact = g.qcd.fermion.mobius(U_stout,{
    "mass": 0.6,
    "M5": 1.0,
    "b": 1.5,
    "c": 0.5,
    "Ls": 12,
    "boundary_phases": [1.0, 1.0, 1.0, -1.0],
})
```

Performance

Benchmark results committed to github

`https://github.com/lehner/gpt/tree/master/benchmarks/reference`

master ▾ gpt / benchmarks / reference /

lehner supermuc-ng timing ··· on Apr 1 History

..	
bnl_knl	8 months ago
juron	8 months ago
juwels_booster	2 months ago
lrz_supermuc_ng	last month
qpace3	8 months ago
qpace4	4 months ago
stampede2_knl	6 months ago
summit	8 months ago

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
7	**JUWELS Booster Module** - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764

Results available for GPU and CPU architectures. In the following, focus on Juwels booster (NVIDIA A100) and QPace4 (A64FX, same as Fugaku).

Juwels Booster (node has 4× A100-40GB): Single-node domain-wall fermion $\not{D}$ operator

```
=============================================
              Initialized GPT
     Copyright (C) 2020 Christoph Lehner
=============================================
GPT :       1.543473 s :
                       : DWF Dslash Benchmark with
                       :     fdimensions  : [64, 32, 32, 32]
                       :     precision    : single
                       :     Ls           : 12
                       :
GPT :       7.958636 s : 1000 applications of Dhop
                       :     Time to complete            : 2.93 s
                       :     Total performance           : 11325.46 GFlops/s
                       :     Effective memory bandwidth  : 7824.86 GB/s
GPT :       7.959499 s :
                       : DWF Dslash Benchmark with
                       :     fdimensions  : [64, 32, 32, 32]
                       :     precision    : double
                       :     Ls           : 12
                       :
GPT :      17.420620 s : 1000 applications of Dhop
                       :     Time to complete            : 5.78 s
                       :     Total performance           : 5749.77 GFlops/s
                       :     Effective memory bandwidth  : 7945.14 GB/s
=============================================
               Finalized GPT
=============================================
```

Compare to HBM bandwidth of 1,555 GB/s per GPU

QPace4 (node has one A64FX): Single-node domain-wall fermion $\not{D}$ operator

```
=============================================
              Initialized GPT
    Copyright (C) 2020 Christoph Lehner
=============================================
GPT :       0.265714 s :
                       : DWF Dslash Benchmark with
                       :     fdimensions  : [24, 24, 24, 24]
                       :     precision    : single
                       :     Ls           : 8
                       :
GPT :      20.218240 s : 1000 applications of Dhop
                       :     Time to complete            : 3.67 s
                       :     Total performance           : 954.90 GFlops/s
                       :     Effective memory bandwidth  : 677.11 GB/s
GPT :      20.218842 s :
                       : DWF Dslash Benchmark with
                       :     fdimensions  : [24, 24, 24, 24]
                       :     precision    : double
                       :     Ls           : 8
                       :
GPT :      45.245379 s : 1000 applications of Dhop
                       :     Time to complete            : 7.36 s
                       :     Total performance           : 475.80 GFlops/s
                       :     Effective memory bandwidth  : 674.77 GB/s
=============================================
               Finalized GPT
=============================================
```

Compare to HBM bandwidth of 1,000 GB/s per A64FX

Juwels Booster (node has 4× A100-40GB): Single-node site-local matrix products

```
============================================
           Initialized GPT
     Copyright (C) 2020 Christoph Lehner
============================================
GPT :       1.589357 s :
                       : Matrix Multiply Benchmark with
                       :     fdimensions  : [48, 48, 48, 128]
                       :     precision    : single
                       :
GPT :      10.985099 s : 10 matrix_multiply
                       :     Object type                : ot_matrix_color(3)
                       :     Time to complete           : 0.0058 s
                       :     Effective memory bandwidth  : 5271.36 GB/s
                       :
GPT :      16.689329 s : 10 matrix_multiply
                       :     Object type                : ot_matrix_spin(4)
                       :     Time to complete           : 0.01 s
                       :     Effective memory bandwidth  : 5333.21 GB/s
                       :
GPT :      62.092583 s : 10 matrix_multiply
                       :     Object type                : ot_matrix_spin_color(4,3)
                       :     Time to complete           : 0.097 s
                       :     Effective memory bandwidth  : 5057.37 GB/s
                       :
```

```
GPT :      62.262581 s :
                       : Matrix Multiply Benchmark with
                       :     fdimensions  : [48, 48, 48, 128]
                       :     precision    : double
                       :
GPT :      72.003471 s : 10 matrix_multiply
                       :     Object type                : ot_matrix_color(3)
                       :     Time to complete           : 0.012 s
                       :     Effective memory bandwidth  : 5264.01 GB/s
                       :
GPT :      78.174681 s : 10 matrix_multiply
                       :     Object type                : ot_matrix_spin(4)
                       :     Time to complete           : 0.02 s
                       :     Effective memory bandwidth  : 5439.91 GB/s
                       :
GPT :     128.232979 s : 10 matrix_multiply
                       :     Object type                : ot_matrix_spin_color(4,3)
                       :     Time to complete           : 0.22 s
                       :     Effective memory bandwidth  : 4416.45 GB/s
                       :
============================================
             Finalized GPT
============================================
```

Compare to HBM bandwidth of 1,555 GB/s per GPU

Juwels Booster (node has 4$\times$ A100-40GB): Inner product (reduction)

```
GPT :      28.406798 s : 100 rank_inner_product
                       :     Object type                : ot_vector_singlet(12)
                       :     Block                      : 4 x 4
                       :     Data resides in            : accelerator
                       :     Performed on               : accelerator
                       :     Time to complete           : 0.13 s
                       :     Effective memory bandwidth  : 4827.16 GB/s
                       :
                       :     rip: timing: unprofiled           = 0.000000e+00 s (=   0.00 %)
                       : rip: timing: rip: view            = 9.706020e-04 s (=   0.70 %)
                       : rip: timing: rip: loop            = 1.369879e-01 s (=  99.30 %)
                       : rip: timing: total                = 1.379585e-01 s (= 100.00 %)
                       :
```

Compare to HBM bandwidth of 1,555 GB/s per GPU

Performance summary

Machine	Operation	Performance	Bandwidth
Booster	$\not{D}$	12 TF/s	7.8 TB/s
Booster	ColorMatrix $\times$		5.2 TB/s
Booster	SpinColorMatrix $\times$		5.1 TB/s
Booster	SpinColorVector $\langle\cdot,\cdot\rangle$		4.8 TB/s
QPace4	$\not{D}$	0.95 TF/s	0.68 TB/s
SuperMUC-NG	$\not{D}$	0.72 TF/s	0.51 TB/s

Single-node SP performance of Wilson $\not{D}$ and linear algebra on Juwels Booster (4xA100, HBM BW 1.6 TB/s per A100), Qpace4 (A64FX, HBM BW of 1 TB/s per node), and the SuperMUC-NG (Skylake 8174). The $\not{D}$ performance is inherited from Grid, the linear algebra performance is based on cgpt.

Status of project

Production use:

- GPT is used right now:
  - on Summit/Booster and CPU based clusters for RBC/UKQCD lattice QCD (g-2) production running
  - on QPACE4, KNL, and Skylake machines (BNL/Stampede2/SuperMUC-NG) for DWF B physics projects (collaboration with Stefan Meinel)
  - for a Wilson-Clover baryon charm physics run on Booster (PI: Collins)
- DWF-projects-specific code is fully tuned, Wilson-Clover-specific code still being optimized

The machine learning module

Example: train simple feed-forward network

In [ ]:
```
import gpt as g
grid = g.grid([4, 4, 4], g.double)
rng = g.random("test")

# network and training data
n = g.ml.network.feed_forward([g.ml.layer.nearest_neighbor(grid)] * 2)
training_input = [rng.uniform_real(g.complex(grid)) for i in range(2)]
training_output = [rng.uniform_real(g.complex(grid)) for i in range(2)]

# cost functional
c = n.cost(training_input, training_output)

# train network
W = n.random_weights(rng)
gd = g.algorithms.optimize.gradient_descent
gd(maxiter=4000, eps=1e-4, step=0.2)(c)(W, W)
```

The quantum computing module

Example: create and measure a 5-qubit bell state

```python
import gpt as g
from gpt.qis.gate import *

rng = g.random("qis_test")

# initial state with 5 qubits, stored in double-precision
st = g.qis.backends.dynamic.state(rng, 5, precision=g.double)

g.message("Initial state:\n",st)

# prepare Bell-type state
st = (H(0) | CNOT(0,1) | CNOT(0,2) | CNOT(0,3) | CNOT(0,4)) * st

g.message("Bell-type state:\n",st)

# measure
st = M() * st

g.message("After single measurement:\n",st)
g.message("Classically measured bits:\n",st.classical_bit)
```

```
GPT :     197.943668 s : Initial state:
                       :   + (1+0j) |00000>
GPT :     197.949198 s : Bell-type state:
                       :   + (0.7071067811865475+0j) |00000>
                       :  + (0.7071067811865475+0j) |11111>
GPT :     197.951478 s : After single measurement:
                       :   + (1+0j) |11111>
GPT :     197.952545 s : Classically measured bits:
                       :  [1, 1, 1, 1, 1]
```

How to use GPT?

https://github.com/lehner/gpt

Quick Start

The fastest way to try GPT is to install Docker, start a Jupyter notebook server with the latest GPT version by running

```
docker run --rm -p 8888:8888 gptdev/notebook
```

and then open the shown link `http://127.0.0.1:8888/?token=<token>` in a browser. You should see the tutorials folder pre-installed.

The docker images are automatically generated for each version that passes the CI interface.

CI currently has test coverage of 96%, running on each pushed commit.

More details - The QIS module

Example: create and measure a 5-qubit bell state

```
import gpt as g
from gpt.qis.gate import *

rng = g.random("qis_test")

# initial state with 5 qubits, stored in double-precision
st = g.qis.backends.dynamic.state(rng, 5, precision=g.double)

g.message("Initial state:\n",st)

# prepare Bell-type state
st = (H(0) | CNOT(0,1) | CNOT(0,2) | CNOT(0,3) | CNOT(0,4)) * st

g.message("Bell-type state:\n",st)

# measure
st = M() * st

g.message("After single measurement:\n",st)
g.message("Classically measured bits:\n",st.classical_bit)
```

```
GPT :     197.943668 s : Initial state:
                       :   + (1+0j) |00000>
GPT :     197.949198 s : Bell-type state:
                       :   + (0.7071067811865475+0j) |00000>
                       :  + (0.7071067811865475+0j) |11111>
GPT :     197.951478 s : After single measurement:
                       :   + (1+0j) |11111>
GPT :     197.952545 s : Classically measured bits:
                       :  [1, 1, 1, 1, 1]
```

Universal set of gates implemented; dynamic memory layout

Implementation of a universal set of gates

Need:

- A bit flipped vector $\Psi_i \equiv X_i \Psi$ (i.e. a NOT gate X_i)
- A projector $P_i^{(1)}$ to the subspace with qubit i in $|1\rangle$ state (and the corresponding $P_i^{(0)} = 1 - P_i^{(1)}$)

Then:

$$R_\phi^{(i)} = P_i^{(0)} + e^{i\phi} P_i^{(1)}\,, \tag{6}$$

$$H^{(i)} = \frac{1}{\sqrt{2}} \left(P_i^{(0)}(X_i + 1) + P_i^{(1)}(X_i - 1) \right)\,, \tag{7}$$

$$\mathrm{CNOT}^{(i,j)} = \frac{1}{\sqrt{2}} \left(P_i^{(1)} X_j + P_i^{(0)} \right)\,. \tag{8}$$

Implementation of $R_\phi^{(i)}$

$$R_\phi^{(i)} = P_i^{(0)} + e^{i\phi} P_i^{(1)} \tag{9}$$

```
def R_z(self, i, phi):
    phase_one = np.exp(1j * phi)
    g.bilinear_combination(
        [self.lattice],
        [
            self.bit_map.zero_mask[self.bit_permutation[i]],
            self.bit_map.one_mask[self.bit_permutation[i]],
        ],
        [self.lattice],
        [[1.0, phase_one]],
        [[0, 1]],
        [[0, 0]],
    )
```

Implementation of $H^{(i)}$

$$H^{(i)} = \frac{1}{\sqrt{2}} \left(P_i^{(0)}(X_i + 1) + P_i^{(1)}(X_i - 1) \right) \tag{10}$$

```
def H(self, i):
    bfl = self.bit_flipped_lattice(i)
    nrm = 1.0 / 2.0 ** 0.5
    g.bilinear_combination(
        [self.lattice],
        [
            self.bit_map.zero_mask[self.bit_permutation[i]],
            self.bit_map.one_mask[self.bit_permutation[i]],
        ],
        [self.lattice, bfl],
        [[nrm, nrm, -nrm, nrm]],
        [[0, 0, 1, 1]],
        [[0, 1, 0, 1]],
    )
```

Implementation of $\mathrm{CNOT}^{(i,j)}$

$$\mathrm{CNOT}^{(i,j)} = \frac{1}{\sqrt{2}} \left(P_i^{(1)} X_j + P_i^{(0)} \right) \tag{11}$$

```python
def CNOT(self, control, target):
    assert control != target
    bfl = self.bit_flipped_lattice(target)
    g.bilinear_combination(
        [self.lattice],
        [
            self.bit_map.zero_mask[self.bit_permutation[control]],
            self.bit_map.one_mask[self.bit_permutation[control]],
        ],
        [self.lattice, bfl],
        [[1.0, 1.0]],
        [[0, 1]],
        [[0, 1]],
    )
```

Probability

Probability to measure qubit i in $|1\rangle$ is a reduction:

```
def probability(self, i):
    return g.norm2(self.lattice * self.bit_map.one_mask[self.bit_permutation[i]])
```