SimuNova

Portfolio Optimization on HPC platforms

Peter Gottschling, Cornelius Steinhardt and Theodore E. Omtzigt

SimuNova UG and Stillwater, Inc

- Goal: Optimizing stock portfolio for minimal variance
- Use: Non-linear optimization
- Realize: On different platforms with one single implementation
  - Readability through natural math notation
  - High-performance adaptive code generation

- For a given set of stocks we search a portfolio such that:
  - The overall yield on shares is a specified r and
  - The variance of the portfolio is minimal.
- The stocks are characterized by:
  - A vector $\vec{\mathrm{r}}$ of expected rates of return and
  - A matrix σ of covariances between any pair of stocks.
- Mathematically spoken, we search a stock mixture π with:

$$\langle \pi, \vec{1} \rangle = 1 \qquad \text{(Sum is 1.)} \tag{1}$$

$$\langle \pi, \vec{\mathrm{r}} \rangle = \mathrm{r} \qquad \text{(Aimed rate of return.)} \tag{2}$$

$$\pi^{\mathrm{T}} \sigma \pi \to \min \tag{3}$$

- The constraints (1) and (2) can be handled by Lagrange functions:

$$\alpha(\langle \pi, \vec{1} \rangle - 1)^2 + \beta(\langle \pi, \vec{\mathrm{r}} \rangle - \mathrm{r})^2 + \pi^{\mathrm{T}} \sigma \pi \to \min \tag{4}$$

with some large α and β.

- Solves non-constraint optimization problems
- Like in equation (4)
- Opposed to regular Newton the second derivative is approximated
  - Already for this approximation numerous methods exist
  - We use Broyden/Fletcher/Goldfarb/Shanno (BFGS)

Algorithm 1: Quasi-Newton

Input: $x^0 \in \mathbb{R}$, $H_0 \in \mathbb{R}^{nxn}$ symmetric positive-definite
$k := 0$
while $\nabla f(x^k) > \varepsilon$ do
    $d^k = -H_k \nabla f(x^k)$
    Find $\alpha > 0$ that holds Wolf's condition
    $x^{k+1} = x^k + \alpha_k d^k$
    $s^k = \alpha_k d^k$
    $y^k = \nabla f(x^{k+1}) - \nabla f(x^k)$
    $\gamma_k = \frac{1}{(y^k)^T s^k}$
    Update H_{k+1} `// e.g. with BFGS`
end

Algorithm 2: Broyen/Fletcher/Goldfarb/Shanno (BFGS)

Input: H^k, s^k, y^k
$\gamma_k = \frac{1}{(y^k)^T s^k}$
return $(I - \gamma_k s^k (y^k)^T) H^k (I - \gamma_k y^k (s^k)^T) + \gamma_k s^k (s^k)^T$

Algorithm 3: BFGS with temporary for less redundancy

Input: H^k, s^k, y^k
$\gamma_k = \frac{1}{(y^k)^T s^k}$
$A = I - \gamma_k s^k (y^k)^T$
return $A \cdot H^k \cdot A^T + \gamma_k s^k (s^k)^T$

```
struct bfgs
{
    template <typename Matrix, typename Vector>
    void operator() (Matrix& H, const Vector& y, const Vector& s)
    {
        Collection<Vector>::value_type gamma= 1 / dot(y,s);
        Matrix A(math::one(H) − gamma ∗ s ∗ trans(y)),
               H2(A ∗ H ∗ trans(A) + gamma ∗ s ∗ trans(s));
        swap(H2, H); // faster than H= H2
    }
};
```

```
template <typename Matrix, typename Vector, typename F, typename Grad,
          typename Step, typename Update, typename Iter>
Vector quasi_newton(Vector& x, F f, Grad grad_f, Step step, Update update, Iter& iter)
{
    typedef typename mtl::Collection<Vector>::value_type value_type;
    Vector d, y, x_k, s;
    Matrix H(size(x), size(x));

    H= 1;
    for (; !iter.finished(two_norm(grad_f(x))); ++iter) {
        d= H * -grad_f(x);
        value_type alpha= step(x, d, f, grad_f);
        x_k= x + alpha * d;
        s= alpha * d;
        y= grad_f(x_k) - grad_f(x);
        update(H, y, s);
        x= x_k;
    }
    return x;
}
```

Algorithm 4: Quasi-Newton

Input: $x^0 \in \mathbb{R}$, $H_0 \in \mathbb{R}^{nxn}$ SPD
$k := 0$
while $\nabla f(x^k) > \varepsilon$ do
- $d^k = -H_k \nabla f(x^k)$
- Find $\alpha > 0$ that holds Wolf
- $x^{k+1} = x^k + \alpha_k d^k$
- $s^k = \alpha_k d^k$
- $y^k = \nabla f(x^{k+1}) - \nabla f(x^k)$
- $\gamma_k = \frac{1}{(y^k)^T s^k}$
- Update H_{k+1}

end

```
Vector quasi_newton(Vector& x, F f, Grad grad_f,
                    Step step, Update update, Iter& iter)
{
    Vector d, y, x_k, s;
    Matrix H(size(x), size(x));

    H= 1;
    for (; !iter.finished(two_norm(grad_f(x))); ++iter)
    {
        d= H * -grad_f(x);
        value_type alpha= step(x, d, f, grad_f);
        x_k= x + alpha * d;
        s= alpha * d;
        y= grad_f(x_k) - grad_f(x);
        update(H, y, s);
        x= x_k;
    }
    return x;
}
```

Today everything can be done nicely on a computer except one thing:
Computing!

- Programmers do not drown in technical details.
- Algorithms are written in purely mathematical notation.
- Programming becomes intuitive (like in Matlab and Mathematica).
- MTL4 provides already an extensive functionality out of the box.

Instead of wasting their time with hacking,
scientist can do science, engineers can do engineering, ...

- MTL4 maximizes performance by compile-optimization
  - Like expression templates
- Meta-Tuning: new technology developped by Peter Gottschling
  - Tuning at compile-time saves from re-implementation
  - See http://simunova.com/node/151#metatuning
- User-transparent interfacing to BLAS, LAPACK, Umfpack, ...
  - Where generic C++ is out-performed by assembler libs.

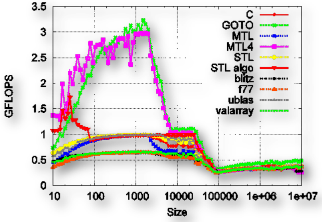

Figure: Comparison of MTL4 with other libraries on AMD Opteron 2GHz: left dot product and right dense matrix product. When BLAS is enabled than MTL4 has identical performance as underlying library.

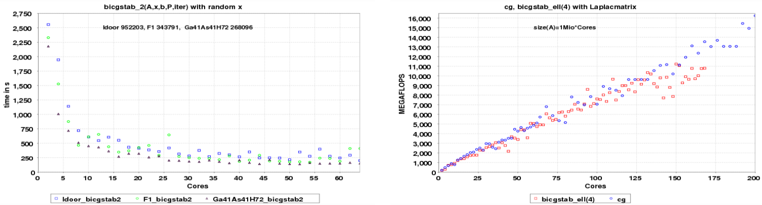

Figure: Speedup (left) and performance gain (right) of different iterative solvers with user-transparent parallelization.

- All shown programs run on entirely different platforms.
- Accordingly, our iterative solvers are implemented once for all platforms.
- All MTL4 applications written on an abstract math level are portable on:
  - Single CPU;
  - Supercomputers and parallel cluster;
  - Graphic processors (under development);
  - GPU cluster by combining the previous two versions
- When new platforms come out, we will port MTL4.
- MTL4 users keep their application codes without changing.
- Using MTL4 makes your software future-proof.

HPC programming will be fun with MTL4!