Workpackage 5: High Performance Mathematical Computing

Clément Pernet

First OpenDreamKit Project review

Brussels, April 26, 2017
High performance mathematical computing

Computer algebra

Typical computation domains:

- \( \mathbb{Z}, \mathbb{Q} \): \( \mapsto \) multiprecision integers
- \( \mathbb{Z}/p\mathbb{Z}, \mathbb{F}_q \): \( \mapsto \) machine ints or floating point, multiprecision
- \( K[X], K^{m\times n}, K[X]^{m\times n} \) for \( K = \mathbb{Z}, \mathbb{Q}, \mathbb{Z}/p\mathbb{Z} \)

High performance computing

- Decades of development for numerical computations
- Still at an early development stage for computer algebra
- Specificites: cannot blindly benefit from numerical HPC experience
Goal: delivering high performance to maths users

- Harnessing modern hardware
  - Parallelisation
    - In-core parallelism (SIMD vectorisation)
    - Multi-core parallelism
    - Distributed computing: clusters, cloud

- Systems:
  - GAP
  - PARI/GP
  - SageMath
  - Singular

- Components:
  - MPIR
  - LinBox
  - NumPy
Goal: delivering high performance to maths users

Harnessing modern hardware \iff parallelisation

- in-core parallelism (SIMD vectorisation)
- multi-core parallelism
- distributed computing: clusters, cloud

Systems:
- GAP
- PARI/GP
- SageMath
- Singular

Components:
- MPIR
- LinBox
- NumPy

Architectures:
- SIMD
- Multicore server
- HPC cluster
- Cloud
Goal: delivering high performance to maths users

Languages

- Computational Maths software uses high level languages (e.g. Python)
- High performance delivered by languages close to the metal (C, assembly)

⇒ compilation, automated optimisation

Systems:
- GAP
- PARI/GP
- SageMath
- Singular

Components:
- MPIR
- LinBox
- NumPy
- Python
- Cython
- Pythran
- C

Architectures:
- SIMD
- Multicore server
- HPC cluster
- Cloud
Main tasks under review for the period

- Task 5.4: Singular
- Task 5.5: MPIR
- Task 5.6: Combinatorics
- Task 5.7: Pythran
- Task 5.8: SunGridEngine in JupyterHub

Progress report on other tasks
Task 5.4: Singular

**Singular**: A computer algebra system for polynomial computations.

- Already has a generic parallelization framework
- Focus on optimising kernel routines for fine grain parallelism

**D5.6**: Quadratic sieving for integer factorization

**D5.7**: Parallelization of matrix fast Fourier Transform
Quadratic Sieving for integer factorization

Problem: Factor an integer \( n \) into prime factors

Role: Crucial in algebraic number theory, arithmetic geometry.

Earlier status: no HPC implementation for large instances:
- only fast code for up to 17 digits,
- only partial sequential implementation for large numbers
D5.6: Quadratic Sieving for integer factorization

Achievements

- Completed and debugged implementation of large prime variant
- Parallelised sieving component of implementation using OpenMP
- Experimented with a parallel implementation of Block Wiedemann algorithm

Results

- Now modern, robust, parallel code for numbers in 17–90 digit range
D5.6: Quadratic Sieving for integer factorization

Achievements

- Completed and debugged implementation of large prime variant
- Parallelised sieving component of implementation using OpenMP
- Experimented with a parallel implementation of Block Wiedemann algorithm

Results

- Now modern, robust, parallel code for numbers in 17–90 digit range
- Significantly faster on small multicore machines

Table: Speedup for 4 cores (c/f single core):

<table>
<thead>
<tr>
<th>Digits</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.1×</td>
<td>1.76×</td>
<td>1.55×</td>
<td>2.69×</td>
<td>2.80×</td>
</tr>
</tbody>
</table>
D5.7: Parallelise and assembly optimise FFT

FFT: Fast Fourier Transform over $\mathbb{Z}/p\mathbb{Z}$

- Among the top 10 most important algorithms
- Key to fast arithmetic (integers, polynomials)
- Difficult to optimise: high memory bandwidth requirement

Earlier status:
- World leading sequential code in MPIR and FLINT;
- No parallel code.
D5.7: Parallelise and assembly optimise FFT

Achievements

▶ Parallelised Matrix Fourier implementation using OpenMP
▶ Assembly optimised butterfly operations in MPIR

Results:

▶ \( \approx 15\% \) speedup on Intel Haswell
▶ \( \approx 20\% \) speedup on Intel Skylake
▶ Significant speedups on multicore machines

Table: Speedup of large integer multiplication on 4/8 cores:

<table>
<thead>
<tr>
<th>Digits</th>
<th>3M</th>
<th>10M</th>
<th>35M</th>
<th>125M</th>
<th>700M</th>
<th>3.3B</th>
<th>14B</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 cores</td>
<td>1.35×</td>
<td>2.67×</td>
<td>2.92×</td>
<td>2.92×</td>
<td>3.01×</td>
<td>2.95×</td>
<td>3.32×</td>
</tr>
<tr>
<td>8 cores</td>
<td>1.35×</td>
<td>3.56×</td>
<td>4.22×</td>
<td>4.36×</td>
<td>4.50×</td>
<td>4.31×</td>
<td>5.49×</td>
</tr>
</tbody>
</table>
MPIR : a library for big integer arithmetic

- Bignum operations: fundamental across all of computer algebra

D5.5: Assembly superoptimisation

- MPIR contains assembly language routines for bignum operations
  - hand optimised for every new microprocessor architecture
  - \( \approx 3 - 6 \) months of work for each architecture
- Superoptimisation: rearranges instructions to get optimal ordering

Earlier status:
- No assembly code for recent (> 2012) Intel and AMD chips (Bulldozer, Haswell, Skylake, ... )
D5.5: Assembly superoptimisation

Achievements

- A new assembly superoptimiser supporting recent instruction sets
- Superoptimised handwritten assembly code for Haswell and Skylake
- Hand picked faster assembly code for Bulldozer from existing implementations

Results:

- Sped up basic arithmetic operations for Bulldozer, Skylake and Haswell
- Noticeable speedups for bignum arithmetic for all size ranges

<table>
<thead>
<tr>
<th>Op</th>
<th>Mul (s)</th>
<th>Mul (m)</th>
<th>Mul (b)</th>
<th>GCD (s)</th>
<th>GCD (m)</th>
<th>GCD (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haswell</td>
<td>1.18×</td>
<td>1.27×</td>
<td>1.29×</td>
<td>0.72×</td>
<td>1.45×</td>
<td>1.27×</td>
</tr>
<tr>
<td>Skylake</td>
<td>1.15×</td>
<td>1.20×</td>
<td>1.22×</td>
<td>0.84×</td>
<td>1.65×</td>
<td>1.32×</td>
</tr>
</tbody>
</table>

s = 512 bits, m = 8192 bits, big = 100K bits
Task 5.6: Combinatorics

Perform a **map/reduce** on huge **recursive** datasets.

Large range of intensive applications in combinatorics:

- Test a conjecture: i.e. find an element of $S$ satisfying a specific property
- Count/list the elements of $S$ having this property
Task 5.6: Combinatorics

Perform a map/reduce on huge recursive datasets.

Large range of intensive applications in combinatorics:

- Test a conjecture: i.e. find an element of $S$ satisfying a specific property
- Count/list the elements of $S$ having this property

Specificities of combinatorics:

- Sets often *don’t fit in the computer’s memory / disks* and are enumerated *on the fly* (example of value: $10^{17}$ bytes).
- *Embarassingly parallel*, if the set is flat (a list, a file, stored on a disk).
- Recursive data-structures may be *heavily unbalanced*
A Challenge: The tree of numerical semigroups
A Challenge: The tree of numerical semigroups

Need for

- an efficient load balancing algorithm.
- a high level task parallelization framework.
Work-Stealing System Architecture

A Python implementation

- Work stealing algorithm (Leiserson-Blumofe / Cilk)
- Easy to use, easy to call from SageMath
- Already, a dozen use cases
- Scale well with the number of CPU cores
- Reasonably efficient (knowing that this is Python code).

<table>
<thead>
<tr>
<th># processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>250</td>
<td>161</td>
<td>103</td>
<td>87</td>
</tr>
</tbody>
</table>

References

- Trac Ticket 13580 http://trac.sagemath.org/ticket/13580
- Exploring the Tree of Numerical Semigroups J. Fromentin and F. Hivert
**Pythran**: a NumPy-centric Python to C compiler

- Many high level VREs rely on the Python language
- High performance is most often achieved by the C language
Pythran: a NumPy-centric Python to C compiler

- Many high level VREs rely on the Python language
- High performance is most often achieved by the C language
- Python to C compilers:
  - Cython: general purpose
  - Pythran: narrower scope, better at optimising Numpy code (Linear algebra)
Pythran: a NumPy-centric Python to C compiler

- Many high level VREs rely on the Python language
- High performance is most often achieved by the C language
- Python to C compilers:
  - Cython: general purpose
  - Pythran: narrower scope, better at optimising Numpy code (Linear algebra)

Goal: Implement the convergence

D5.4 Improve Pythran typing system
D5.2 Make Cython use Pythran backend to optimise Numpy code
D5.2: Make Cython use Pythran backend for NumPy code

```python
import numpy

ciimport numpy
def float_comp(a, b):
    return numpy.sum(numpy.sqrt(a*a+b*b))
```
D5.2: Make Cython use Pythran backend for NumPy code

```python
def harris(numpy.ndarray[numpy.float_t, ndim=2] I):
    cdef int m = I.shape[0]
    cdef int n = I.shape[1]
    cdef numpy.ndarray[numpy.float_t, ndim=2] dx = (I[1:,:]-I[:m-1,:])[1:,1:]
    cdef numpy.ndarray[numpy.float_t, ndim=2] dy = (I[:1,:]-I[:,:n-1])[1:,1:]
    cdef numpy.ndarray[numpy.float_t, ndim=2] A = dx*dx
    cdef numpy.ndarray[numpy.float_t, ndim=2] B = dy*dy
    cdef numpy.ndarray[numpy.float_t, ndim=2] C = dx*dy
    cdef numpy.ndarray[numpy.float_t, ndim=2] tr = A + B
    cdef numpy.ndarray[numpy.float_t, ndim=2] det = A * B - C * C
    return det - tr * tr
```

Clément Pernet: Workpackage 5  17  Brussels, April 26, 2017
Task 5.8: SunGridEngine integration in JupyterHub

Access to big compute

- Traditional access to supercomputers is difficult
- Notebooks are easy but run on laptops or desktops
- We need a way to connect notebooks to supercomputers

Sun Grid Engine

A job scheduler for Academic HPC Clusters

- Controls how resources are allocated to researchers
- One of the most popular schedulers

Achievements: D5.3

- Developed software to run Jupyter notebooks on supercomputers
- Users don’t need to know details. They just log in.
- Demonstration install at University of Sheffield
Main tasks under review for the period

Task 5.4: Singular
Task 5.5: MPIR
Task 5.6: Combinatorics
Task 5.7: Pythran
Task 5.8: SunGridEngine in JupyterHub

Progress report on other tasks
Progress report on other tasks

T5.1: PARI
▶ Generic parallelization engine is now mature, released (D5.10, due M24)

T5.2: GAP
▶ 6 releases were published integrating contributions of D3.11 and D5.15
▶ Build system refactoring for integration of HPC GAP

T5.3: LinBox
▶ Algorithmic advances (5 articles) on linear algebra and verified computing
▶ Software releases and integration into SageMath
WP5 highlights

Sites involved: UPSud, CNRS, UJF, UNIKL, USFD, USTAN, Logilab
Workforce: 49.58 PM (consumed) / 200 PM (total)
Delivered: 7 deliverables

- Optimized parallel kernels: FFT, factorization, bignum arithmetic.
- New assembly superoptimizer supporting last generation CPUs
- Workstealing based task parallelization for combinatorics exploration
- Cython can use Pythran backend to compile Numpy Code
- Jupyter can be run on Cluster nodes using SunGridEngine scheduler