Run, Stencil, Run!
HPC Productivity Studies in the Classroom

6th Conference on Partitioned Global Address Space Programming Models
October 10-12, 2012
Santa Barbara, California, USA

Helmar Burkhart\textsuperscript{1}, Matthias Christen\textsuperscript{2}, Max Rietmann\textsuperscript{2}, Madan Sathe\textsuperscript{1}, Olaf Schenk\textsuperscript{2}

\textsuperscript{1} Department of Mathematics and Computer Science, University of Basel, Switzerland
\textsuperscript{2} Faculty of Informatics, University of Lugano, Switzerland
P-Challenges 2012 and Beyond

- **Performance:** Preparations for Exascale.
- **Power:** Computation cheap, data movement expensive.
- **Parallelism:** > $10^6$ fine-granular schedules.
- **Programmability:** GPLs & DSLs, Libraries.
- **Portability:** Late System-binding. Auto-tuning.

**Productivity:** What is it? How to measure?
Where to Find a Crowd for Productivity Studies?

- Do **not** want experienced HPC programmers who have already personal favourites.
  → Newcomers open to all parallel programming models.
- Do **not** want pure HPC enthusiasts.
  → Good mix of users: Some freaks, but majority wants to do just their job.

MSc students in Computer Science are a reasonable approximation for such a crowd!

Before my course!
Part I: Classroom Studies

CS311 High-Performance Computing
- Lectures: Algorithms, Architecture, and Programming
- Lab work mandatory
- 6 ECTS Credit points (=180h work).

Audience
- Graduate-level course
- Computer Science and Computational Science master students, a few PhD students.
- 15 participants in 2012

Challenge: How do make students familiar both with state-of-the-art and next-generation HPC?
Stencil Motif is Motivating

- Geo-Physics
- Meteorology
- Computational Medicine
- Engineering
- Image Processing
- …
Stencil Motif is a Challenge

Arithmetic Intensity := Flops / Transferred Data

Stencils
Sparse Linear Algebra
BLAS 1
BLAS 2

FFT
Lattice Methods

Particle methods
Dense Linear Algebra (BLAS3)

\( \mathcal{O}(1) \)
\( \mathcal{O}(\log n) \)
\( \mathcal{O}(n) \)

Low arithmetic intensity
⇒ memory bandwidth bound

High arithmetic intensity
⇒ processor bound
Solve the classical wave equation using finite differences.

\[ \frac{\partial^2 u}{\partial t^2} - c^2 \nabla^2 u = 0 \quad \text{in } \Omega \]

\[ u \equiv 0 \quad \text{on } \partial \Omega \]

\[ u(x, y, z, 0) = \sin(2\pi x) \sin(2\pi y) \sin(2\pi z) \]
Algorithm 1 Pseudo-code for the iterative stencil computation

Require: Arrays $u^-, u^0$; Stencil operator $\nabla_h^2$
Ensure: Array $u^+$

1: for $t \leftarrow 1 \ldots t_{\max}$ do \hspace{1cm} \triangleright Iterate over the time domain
2: \hspace{1cm} \triangleright Iterate over the discrete domain, the index set $\Omega_h$
3: for $(i, j, k) \in \Omega_h$ do
4: \hspace{1cm} \triangleright Compute the next time step at $(i, j, k)$
5: $u^+_{i,j,k} \leftarrow 2u^0_{i,j,k} - u^-_{i,j,k} + \Delta t^2 c^2 \nabla_h^2 u^0_{i,j,k}$
6: end for
7: $u^- \leftarrow u^0$; $u^0 \leftarrow u^+$ \hspace{1cm} \triangleright Rotate arrays (copy semantics)
8: end for

Parameter Settings:
- $200^3$ grid points
- 19 Flops per grid point
- 100 time steps
Target 1: CPU System

- 4 quad-core Intel Xeon E7420 ("Dunnington") CPUs, 2.1 GHz
- 32 KB L1 data cache
- 3 MB unified L2 cache, shared among 2 cores
- 8 MB L3 cache, shared among 4 cores
- 128 GB DDR3 RAM
- 8.3 GB/s BW (measured)
Target 2: GPU System

- 14 "streaming multiprocessors", 448 CUDA cores, 1.15 GHz
- 64 KB shared memory per streaming multiprocessor
- 3 GB GDDR5 RAM
- 79 GB/s BW (measured)
Programming Models to Explore

• Java Concurrency  
  Mainstream language going parallel
• Chapel  
  New high-level GPL & DSL
• PATUS
• OpenMP  
  State-of-the-art industry use
• MPI
• CUDA
class Calculator implements Runnable {
    private int m_nStart;
    private int m_nEnd;

    public Calculator(int nStart, int nEnd) {
        m_nStart = nStart; m_nEnd = nEnd;
    }

    @Override public void run() {
        for (int k = m_nStart; k < m_nEnd; k++)
            for (int j = 2; j < y_max - 2; j++)
                for (int i = 2; i < x_max - 2; i++)
                    u_1[k][j][i]=2*u_0[k][j][i]-u_m1[k][j][i]+...
    }
}
public static void main (String[] args) {
    final int nNumThreads = Runtime.getRuntime().availableProcessors();
    final ExecutorService executor = Executors.newFixedThreadPool(nNumThreads);
    final int nPlanesPerThread = (z_max-4) / nNumThreads;
    for (int t = 0; t < t_max; t++) {
        List<Future<?>> listFutures = new ArrayList<>();
        // submit parallel tasks
        for (int i = 0; i < nNumThreads; i++)
            listFutures.add(executor.submit(
                new Calculator(i*nPlanesPerThread+2, (i+1)*nPlanesPerThread+2)));
        for (Future<?> future : listFutures) { // synchronize
            try { future.get (); } catch ... }
        float[][][] tmp = u_m1; u_m1 = u_0; u_0 = u_1; u_1 = tmp;
    }
    executor.shutdown();
}
# Questionnaire

<table>
<thead>
<tr>
<th>#Threads</th>
<th>Time [seconds]</th>
<th>GFlop/s</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Productivity:**

- Time spent on parallelization: [ ] hours
- Lines of source code: sequential: [ ] parallel: [ ]
- Learning curve: ○ easy ○ medium ○ hard
- Difficulties during programming:
  - Source of errors (index calculation, parallelization, ...)
  - Kind of errors (conceptional understanding, syntax, programming, ...)

15
Data Collected from Questionnaires

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Working Hours</th>
<th>Parallel Overhead</th>
<th>Best Performance [GFlop/s]</th>
<th>Lines of Code</th>
<th>Learning Curve [1=easy, 3=hard]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>5.14</td>
<td>35%</td>
<td>5.98</td>
<td>267</td>
<td>2.4</td>
</tr>
<tr>
<td>Chapel</td>
<td>4.83</td>
<td>0%</td>
<td>3.23</td>
<td>120</td>
<td>1.8</td>
</tr>
<tr>
<td>OpenMP</td>
<td>2.58</td>
<td>4%</td>
<td>6.66</td>
<td>197</td>
<td>2.0</td>
</tr>
<tr>
<td>MPI</td>
<td>18.00</td>
<td>32%</td>
<td>6.06</td>
<td>296</td>
<td>3.0</td>
</tr>
<tr>
<td>PATUS</td>
<td>2.60</td>
<td>N/A</td>
<td>7.72</td>
<td>22</td>
<td>1.5</td>
</tr>
<tr>
<td>CUDA</td>
<td>8.50</td>
<td>N/A</td>
<td>35.70</td>
<td>183</td>
<td>2.25</td>
</tr>
</tbody>
</table>
Performance Benchmark Results

- Java
- Chapel
- OpenMP
- MPI
- Patus

Single Precision GFlop/s
Productivity (1)

Coding Productivity =

Lines of Code / Working hours
Performance

Productivity =

Performance Achieved / Working hours
Part II: Post-Classroom PGAS Experiments

- How do different PGAS languages compare?
- What performance gain is achievable by hand-made PGAS program tuning?
- Same problem, same target system.
- Advanced HPC programmers / language implementors may find more optimization parts!
const
    Domain = [ 0..x_max-1, 0..y_max-1, 0..z_max-1 ],
    Inner = Domain.expand(-2),
    Left1 = Inner.translate(-1, 0, 0),
    Right1 = Inner.translate(1, 0, 0), ...
var u_m1, u_0, u_1: [ Domain ] real(32);
for t in 1 .. t_max {
    u_1[Inner] = c1 * u_0[Inner] - u_m1[Inner] +
        dt_dx_sq * (c2 * u_0[Inner] +
            c3 * (u_0[Left1] + u_0[Right1] + u_0[Back1] + ...) -
            c4 * (u_0[Left2] + u_0[Right2] + u_0[Back2] + ...));
    u_m1 = u_0; u_0 = u_1;
}
Chapel 2: Subdomain and Reduction

```chapel
const Stencil: sparse subdomain([-2..2, -2..2, -2..2]) = (0,0,0), (-2,0,0), (-1,0,0), (1,0,0), (2,0,0), ...);
var c: [Stencil] real(32);
c[(0, 0, 0)] = c1; c[(-1, 0, 0)] = c2; ...
for t in 1 .. t_max {
    forall i in Inner do
        u_1(i)=(+ reduce [j in Stencil] c(j)*u_0(i+j)) - u_m1(i);
    u_m1 = u_0; u_0 = u_1;
}
```
const
Domain: domain(3) dmapped Block (
    boundingBox = [0..x_max-1, 0..y_max-1, 0..z_max-1]) =
    [ 0..x_max-1, 0..y_max-1, 0..z_max-1 ],
Inner: domain(3) dmapped Block ( 
    boundingBox = [0..x_max-1, 0..y_max-1, 0..z_max-1]) = 
Domain.expand(-2);
for t in 1 .. t_max {
    forall z in 2 .. z_max-3 do
        for y in 2 .. y_max-3 do
            for x in 2 .. x_max-3 do
                u_1(x,y,z) = c1 * u_0(x,y,z) - u_m1(x,y,z) +
                            dt_dx_sq * (c2 * u_0(x,y,z) +
                                 c3 * (u_0(x+1,y,z) + u_0(x-1,y,z) + ...));
                u_m1 = u_0; u_0 = u_1;
}
Chapel 5: Zippered Iterator

```chapel
for t in 1 .. t_max {
    forall (x,y,z) in Inner do
        u_1(x,y,z) = c1 * u_0(x,y,z) - u_m1(x,y,z) + ...
        u_m1 = u_0; u_0 = u_1;
}
```
Chapel 6: 4D-Grid

```plaintext
const
    Inner = [ 1..x_max, 1..y_max, 1..z_max, 0..2 ],
    Domain = DomainInner.expand (2, 2, 2, 0);
var u: [ Domain ] real(32);
for t in 1 .. t_max {
    forall (x,y,z) in Inner do
        u(x,y,z,t-1) = c1 * u(x,y,z,t-0) - u(x,y,z,t_m1) + ... 
    t_1 += 1;  if (t_1 > 2) then t_1 -= 3; // etc. (t_0, t_m1)
}
```
for t in 1 .. t_max/3 {
    forall (x,y,z) in Inner do
        u_1(x,y,z) = c1 * u_0(x,y,z) - u_m1(x,y,z) + ...
    forall (x,y,z) in Inner do
        u_m1(x,y,z) = c1 * u_1(x,y,z) - u_0(x,y,z) + ...
    forall (x,y,z) in Inner do
        u_0(x,y,z) = c1 * u_m1(x,y,z) - u_1(x,y,z) + ...
}
Chapel Performance for Variants

- **Domain shifting**
- **Triply nested “forall”**
- **“forall” with domain map**
- **“forall” zippered**
- **4-D grid**
- **“forall” with unrolling**

The diagram compares the performance of various Chapel variants in Single Precision GFlop/s for different grid sizes (1, 2, 4, 8, 16).
PGAS Language Comparisons

Performances of PGAS Languages

- Chapel
- UPC
- CAF

Single Precision GFlop/s

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Basic
- Improved

29
Conclusions

- As parallel processing goes mainstream, enhanced productivity benchmarks are needed.
- Our studies show that PGAS languages already do quite well compared to production models (with much longer history).
- Multi-resolution approach like in Chapel is good for algorithmic creativity but cost models are needed to predict performance at early stages.
- Performance tuning within the PGAS domain is possible. It is up to the time and money budget how far you go.
- PGAS is the right approach for HPC education.
Outlook

*Run, Stencil, Run* will be continued, a new crowd is waiting:

- Multinode target system
- X10 experiments
- ...

All suggestions how to get improved studies are highly welcome

 Helmar.Burkhart@unibas.ch