Par4All

Open source parallelization for heterogeneous computing
OpenCL & more

Ronan KERYELL\(^3\) (rk@hpc-project.com)

HPC Project

9 Route du Colonel Marcel Moraine\(^1\)
92360 Meudon La Forêt, France

Rond Point Benjamin Franklin\(^2\)
34000 Montpellier, France

Wild Systems, Inc.
5201 Great America Parkway #3241\(^3\)
Santa Clara, CA 95054, USA

25/01/2012
SOLOMON

- Target application: “data reduction, communication, character recognition, optimization, guidance and control, orbit calculations, hydrodynamics, heat flow, diffusion, radar data processing, and numerical weather forecasting”

- Diode + transistor logic in 10-pin TO5 package

POMP & PompC @ LI/ENS 1987–1992

Parallel Processor

- 88100
- 512 KB RAM
- HyperCom

Global exception

FIFO

Scalar Reduction

I/O

Video

Network

Scalar broadcast

Vectorial code

Scalar code

LIW code

Control

RAM

Scalar processor

DACPAV

512 KB RAM

Up to 256

@D

@D

R

G

B

Pomp & PompC @ LI/ENS 1987–1992

- Parallel computer
- Proprietary 3D-torus network
- DEC Alpha 21064 + FPGA
- HyperC (follow-up of PompC @ LI/ENS Ulm)
  - PGAS (Partitioned Global Address Space) language
  - An ancestor of UPC...
- Already on the Saclay Plateau!

Quite simple business model

- Customers need just to rewrite all their code in HyperC
- Difficult entry cost...

- Niche market...
- American subsidiary with dataparallel datamining application acquired by Yahoo! in 1998
- Closed technology lost for customers and... founders

- Parallel computer
- Proprietary 3D-torus network
- DEC Alpha 21064 + FPGA
- HyperC (follow-up of PompC @ LI/ENS Ulm)
  - PGAS (Partitioned Global Address Space) language
  - An ancestor of UPC...
- Already on the Saclay Plateau ! 😊

Quite simple business model

- Customers need just to rewrite all their code in HyperC 😊
- Difficult entry cost... 😞

- Niche market... 😊
- American subsidiary with dataparallel datamining application acquired by Yahoo! in 1998
- Closed technology ⟷ lost for customers and... founders 😞
Present motivations: reinterpreting Moore’s law

The good news 😊

- Number of transistors still increasing
- Memory storage increasing (DRAM, FLASH...)
- Hard disk storage increasing
- Processors (with captors) everywhere
- Network is increasing

The bad news 😞

- Transistors are so small they leak... Static consumption
- Superscalar and cache are less efficient compared to transistor budget
- Storing and moving information is expensive, computing is cheap: change in algorithms...
- Light’s speed has not improved for a while... Hard to reduce latency

- Chips are too big to be globally synchronous at multi GHz 😎
Present motivations: reinterpreting Moore’s law (II)

- pJ and physics become very fashionable
- Power efficiency in $O\left(\frac{1}{f}\right)$
  - Transistors cannot be used at full speed without melting 😒
- I/O and pin counts
  - Huge time and energy cost to move information outside the chip 😌

Parallelism is the only way to go...

Research is just crossing reality!

No one size fit all...

Future will be heterogeneous: GPGPU, Cell, vector/SIMD, FPGA, PIM...

But compilers are always behind... 😃
Off-the-shelf AMD/ATI Radeon HD 7970 GPU

- 4.313 billion 28nm transistors, 352mm²
- 2048 stream processors @ 925 MHz, 3.8 TFLOPS SP, 947 GFLOPS DP
- + External 3 GB GDDR5 memory 5.5 Gt/s, 264 GB/s, 384b GDDR5
- 250 W on board (15 idle), PCI Express 3.0 x16 bus interface
- OpenGL, OpenCL
- ∃ Radeon HD 7990 double chip card

More integration:
- Llano APU (FUSION Accelerated Processing Unit) : x86 multicore + GPU 32nm, OpenCL
Radeon HD 6870 — big picture
Off-the-shelf nVidia Tesla Fermi M2090 & GTX580

- GF110: 3 billion 40nm tr.
- 512 thread processors @ 1300 MHz, 1.3 TFLOPS SP, 666 GFLOPS DP
- + External 6 GB GDDR5 ECC memory 3.7 Gt/s, 177 GB/s. Less if using ECC
- 247 W on board PCI Express 2.1 x16 bus interface
- OpenGL, OpenCL, CUDA
- ∃ GTX590 double chip card
MPPA de Kalray (the French touch!)

- 256 VLIW processors per chip 16 clusters of 16 processors
- Shared memory and NoC with DMA
- 28 nm CMOS technology, \( \approx 5 \) W @ 400 MHz
- FPU 32/64 bits IEEE 754: 205 GFLOPS SP, 1 TOPS 16 bits
- 2\( \times \) 64-bit DDR3 memory controllers for high bandwidth main memory transfers
- 2\( \times \) 40 Gb/s or 8\( \times \) 10 Gb/s Ethernet controller
• 2× 8-lane PCI Express Gen 3
• 4× 4–8-lane Interlaken interfaces for multi-MPPA chip system integration (8 MPPA/PCIe board) or connection to external FPGAs, I/O...
• Linux or bare metal with AccessCore library
• Multi-core compiler (gcc 4.5), simulator, debugger (gdb), profiler
• Eclipse IDE
• Programming from high-level C-based language
• AccessCore library
ARM yourself

- Do some computations where the captors are...
- Smartphone and other sensor networks
- Trade-off between communication energy and inside/remote computations
- Texas Instrument OMAP4470 announced on 2011/06/02
  - 2 ARM Cortex-A9 MPCores @ 1.8GHz with Neon vector instructions
  - 2 ARM Cortex-M3 cores (low-power and real-time responsiveness, multimedia, avoiding to wake up the Cortex-A9...)
  - **SGX544 graphics core with OpenCL 1.1 support**, with 4 USSE2 core @ 384 MHz producing each 4 FMAD/cycle: 12.3 GFLOPS
  - 2D graphics accelerator
  - 3 HD displays and up to QXGA (2048x1536) resolution + stereoscopic 3D
  - Dual-channel, 466 MHz LPDDR2 memory
Current course to have non-x86 servers based on ARM...

Experiments on low power clusters

Think to evaluate power consumption on your application
HPC Project emergence

2006: Time to be back in parallelism!
Yet another start-up... 😊

- People that met ≈ 1990 at the French Parallel Computing military lab SEH/ETCA
- Later became researchers in Computer Science, CINES director and ex-CEA/DAM, venture capital and more: ex-CEO of Thales Computer, HP marketing...
- HPC Project launched in December 2007
- ≈ 30 colleagues in France (Montpellier, Meudon), Canada (Montréal with Parallel Geometry) & USA (Santa Clara, CA)
HPC Project hardware: WildNode from Wild Systems

Through its Wild Systems subsidiary company

- WildNode hardware desktop accelerator
  - Low noise for in-office operation
  - x86 manycore
  - nVidia Tesla GPU Computing
  - Linux & Windows

- WildHive
  - Aggregate 2-4 nodes with 2 possible memory views
    - Distributed memory with Ethernet or InfiniBand

http://www.wild-systems.com
HPC Project software and services

- Parallelize and optimize customer applications, co-branded as a bundle product in a WildNode (e.g. Presagis Stage battle-field simulator, Wild Cruncher for Scilab/...)
- Acceleration software for the WildNode
  - CPU+GPU-accelerated libraries for C/Fortran/Scilab/Scilab/Octave/R
  - Automatic parallelization for Scilab, C, Fortran...
  - Transparent execution on the WildNode
- Par4All automatic parallelization tool
- Remote display software for Windows on the WildNode

HPC consulting

- Optimization and parallelization of applications
- *High Performance*?... not only TOP500-class systems: power-efficiency, embedded systems, green computing...
- Embedded system and application design
- Training in parallel programming (OpenMP, MPI, TBB, CUDA, OpenCL...)

HPC Project — Wild Systems

Par4All @ Paris HPC+GPU meetup
We need software tools

- HPC Project needs tools for its hardware accelerators (*Wild Nodes* from *Wild Systems*) and to parallelize, port & optimize customer applications
- Application development: long-term business $\leadsto$ long-term commitment in a tool that needs to survive to (too fast) technology change
Expressing parallelism?

- Solution libraries
  - Need to fit your application

- New parallel languages
  - Rewrite your applications...

- Extend sequential language with #pragma
  - Nicer transition

- Hide parallelism in object oriented classes
  - Restructure your applications...

- Use magical automatic parallelizer
Automatic parallelization

- Major research failure from the past...
- Untractable in the general case 😞
- Bad sequential programs? GIGO: Garbage In-Garbage Out...
- But technology widely used locally in main compilers
- To use #pragma, // languages or classes: cleaner sequential program or algorithm first...
- ... and then automatic parallelization can often work 😊
- Par4All = automatic parallelization + coding rules
- Often less optimal performance but better time-to-market
Basic Par4All coding rules for good parallelization

- Develop a coding rule manual to help parallelization and sequential quality!
- Par4All parallelizes loop-nests made from Fortran DO or C99 for loops similar to DO-loops
- Same constraints as for-loop accepted in OpenMP standard
- `for ([int] \text{init-expr}; \text{var relational-op b; incr-expr}) \text{statement}`
- Increment and bounds: integer expressions, loop-invariant
- `relational-op` only `<`, `<=`, `>=`, `>`
- Do not modify loop index inside loop body
- Do not use `assert()` or compile with `-DNDEBUG` inside a loop. Assert has potential exit effect
- No `goto` outside the loop, `break`, `continue`
Basic Par4All coding rules for good parallelization (II)

- **No** `exit()`, `longjump()`, `setcontext()`...

- **Data structures**
  - **Pointers**
    - Do not use pointer arithmetics
  - **Arrays**
    - PIPS uses integer polyhedron lattice in analysis, so use affine reference in parallelizable code
    
      1   // Good:
      a[2*i-3+m][3*i-j+6*n]
      
      3   // Bad (polynomial):
      a[2*i*j][m*n-i+j]
    - Do not use linearized arrays

- **Do not use recursion**

- **Prototype of coding rules report on-line on par4all.org**
Bad/good C programming example

Dynamically allocated 2-D array with dynamic size

- **Bad:**

```c
int n, m;
double * t = malloc(sizeof(double)*n*m);
for(int i = 0; i < n; i++)
    for(int j = 0; j < m; m++)
        t[i*n + j] = i + j;
free(t);
```

- **Good:**

```c
double (*t)[n][m] = malloc(sizeof(double (*)[n][m]));
for(int i = 0; i < n; i++)
    for(int j = 0; j < m; m++)
        t[i][j] = i + j;
```

- **Good:**

```c
{
    double t[n][m];
    for(int i = 0; i < n; i++)
        for(int j = 0; j < m; m++)
            t[i][j] = i + j;
}
```

Before parallelization, good sequential programming...
Clean programming for Par4All

- Pointers and global variables are often issues with parallelization.
- OpenMP parallelization is more tolerant, but compilation for heterogeneous computing need more difficult communication analysis.
- Typical pattern to hide nastiness under the carpet:

```c
1 double global_array[N][M];
2 void some_gory_function(float *array, int size) {
    int auto_array[size];
    double (*heap_array)[size][size] = malloc(sizeof(double (*)[size][size]));
    float (*cleaner_array)[size] = (float (*)[size]) array;
    clean_function(size, auto_array, *heap_array, *cleaner_array, global_array);
}

// Here all arrays are C99-style now!
void clean_function(int size, int auto_array[size],
                    double heap_array[size][size],
                    float cleaner_array[size], double global_array[N][M]) {
    // Use no global arrays here
}
```
Parallelisation

p4a matmul.f

generates an OpenMP program in matmul.p4a.f

```fortran
!$omp parallel do private(I, K, X)
C multiply the two square matrices of ones
   DO J = 1, N
!$omp parallel do private(K, X)
      DO I = 1, N
         X = 0
!$omp parallel do reduction(:+X)
         DO K = 1, N
            X = X + A(I,K)*B(K,J)
      ENDDO
!$omp end parallel do
      C(I,J) = X
   ENDDO
!$omp end parallel do
!$omp end parallel do
```

Par4All @ Paris HPC+GPU meetup
Parallelisation with compilation

```bash
p4a matmul.f -o matmul
```
generates an OpenMP program `matmul.p4a.f` that is compiled with `gcc` into `matmul`

CUDA generation with compilation

```bash
p4a --cuda saxpy.c -o s
```
generates a CUDA program that is compiled with `nvcc`

OpenCL generation with compilation

```bash
p4a --opencl saxpy.c -o s
```
Program execution on GPU with CUDA or OpenCL
Basic GPU execution model

A sequential program on a host launches computational-intensive kernels on a GPU

- Allocate storage on the GPU
- Copy-in data from the host to the GPU
- Launch the kernel on the GPU
- The host waits...
- Copy-out the results from the GPU to the host
- Deallocate the storage on the GPU

Generic scheme for other heterogeneous accelerators too
Rely on PIPS

- PIPS (Interprocedural Parallelizer of Scientific Programs): Open Source project from Mines ParisTech... 23-year old! 😊
- Funded by many people (French DoD, Industry & Research Departments, University, CEA, IFP, Onera, ANR (French NSF), European projects, regional research clusters...)
- One of the project that introduced polytope model-based compilation
- ≈ 456 KLOC according to David A. Wheeler’s SLOCCount
- ... but modular and sensible approach to pass through the years
  - ≈300 phases (parsers, analyzers, transformations, optimizers, parallelizers, code generators, pretty-printers...) that can be combined for the right purpose
  - Abstract interpretation
Rely on PIPS (II)

- **Polytope lattice** *(sparse linear algebra)* used for semantics analysis, transformations, code generation... with *approximations* to deal with big programs, not only
- NewGen object description language for language-agnostic automatic generation of methods, persistence, object introspection, visitors, accessors, constructors, XML marshaling for interfacing with external tools...
- Interprocedural *à la make* engine to chain the phases as needed. Lazy construction of resources
- On-going efforts to extend the semantics analysis for C

- Around 15 programmers currently developing in PIPS (Mines ParisTech, HPC Project, IT SudParis, TÉLÉCOM Bretagne, RPI) with public *svn*, Trac, *git*, mailing lists, IRC, Plone, Skype... and use it for many projects
- But still...
  - Huge need of documentation (even if PIPS uses literate programming...)
Rely on PIPS

- Need of industrialization
- Need further communication to increase community size
Current PIPS usage

- Automatic parallelization (Par4All C & Fortran to OpenMP)
- Distributed memory computing with OpenMP-to-MPI translation [STEP project]
- Generic vectorization for SIMD instructions (SSE, VMX, Neon, CUDA, OpenCL...) (SAC project) [SCALOPES]
- Parallelization for embedded systems [SCALOPES, SMECY]
- Compilation for hardware accelerators (Ter@PIX, SPoC, SIMD, FPGA...) [FREIA, SCALOPES]
- High-level hardware accelerators synthesis generation for FPGA [PHRASE, CoMap]
- Reverse engineering & decompiler (reconstruction from binary to C)
- Genetic algorithm-based optimization [Luxembourg university+TB]
- Code instrumentation for performance measures
- GPU with CUDA & OpenCL [TransMedi@, FREIA, OpenGPU]
### Automatic parallelization

**Most fundamental for a parallel execution**

**Finding parallelism!**

<table>
<thead>
<tr>
<th>Several parallelization algorithms are available in PIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>• For example classical Allen &amp; Kennedy use loop distribution more vector-oriented than kernel-oriented (or need later loop-fusion)</td>
</tr>
<tr>
<td>• Coarse grain parallelization based on the independence of array regions used by different loop iterations</td>
</tr>
<tr>
<td>➤ Currently used because generates GPU-friendly coarse-grain parallelism</td>
</tr>
<tr>
<td>➤ Accept complex control code without <em>if-conversion</em></td>
</tr>
</tbody>
</table>
Parallel code \( \leadsto \) Kernel code on GPU

- Need to extract parallel source code into kernel source code: outlining of parallel loop-nests

- Before:

```plaintext
for (i = 1; i <= 499; i++)
    for (j = 1; j <= 499; j++) {
        save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
                        + space[i][j - 1] + space[i][j + 1]);
    }
```

- Outlining of parallel loop-nests
After:

```c
void p4a_kernel_launcher_0(float_t space[SIZE][SIZE], float_t save[SIZE][SIZE]) {
    for (i = 1; i <= 499; i += 1)
        for (j = 1; j <= 499; j += 1)
            p4a_kernel_0(i, j, save, space);
}
```

```c
void p4a_kernel_0(float_t space[SIZE][SIZE], float_t save[SIZE][SIZE], int i, int j) {
    save[i][j] = 0.25*(space[i-1][j]+space[i+1][j] +space[i][j-1]+space[i][j+1]);
}
```
From array regions to GPU memory allocation

Example

```c
for (i = 0; i <= n-1; i += 1)
    for (j = i; j <= n-1; j += 1)
        h_A[i][j] = 1;
```

- Memory accesses are summed up by inference for each statement as regions for array accesses: integer polytope lattice

```c
// < h_A[PHI1][PHI2] - W-EXACT {0 <= PHI1, PHI2+1 <= n, PHI1 <= PHI2} >
for (i = 0; i <= n-1; i += 1)
    for (j = i; j <= n-1; j += 1)
        // < h_A[PHI1][PHI2] - W-EXACT {PHI1=i, i<=PHI2, PHI2+1<=n, 0<=i} >
        h_A[i][j] = 1;
```

- These read/write regions for a kernel are used to allocate with a `cudaMalloc()` in the host code the memory used inside a kernel and to deallocate it later with a `cudaFree()`
Communication generation

More subtle approach

PIPS gives 2 very interesting region types for this purpose

- **In-region** abstracts what really needed by a statement
- **Out-region** abstracts what really produced by a statement to be used later elsewhere

- In-Out regions can directly be translated with CUDA into
  
  ```
  // copy-in
  cudaMemcpy(accel_address, host_address, size, cudaMemcpyHostToDevice)
  
  // copy-out
  cudaMemcpy(host_address, accel_address, size, cudaMemcpyDeviceToHost)
  ```
Loop normalization

- Hardware accelerators use fixed iteration space (thread index starting from 0...)
- Parallel loops: more general iteration space
- Loop normalization

Before

```c
for (i = 1; i < SIZE - 1; i++)
    for (j = 1; j < SIZE - 1; j++) {
        save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
                          + space[i][j - 1] + space[i][j + 1]);
    }
```

After

```c
for (i = 0; i < SIZE - 2; i++)
    for (j = 0; j < SIZE - 2; j++) {
        save[i+1][j+1] = 0.25*(space[i][j + 1] + space[i + 2][j + 1]
                               + space[i + 1][j] + space[i + 1][j + 2]);
    }
```
• Parallel loop nests are compiled into a CUDA kernel wrapper launch
• The kernel wrapper itself gets its virtual processor index with some `blockIdx.x*blockDim.x + threadIdx.x`
• Since only full blocks of threads are executed, if the number of iterations in a given dimension is not a multiple of the `blockDim`, there are incomplete blocks 😞
• An incomplete block means that some index overrun occurs if all the threads of the block are executed ❗️
So we need to generate code such as

```c
void p4a_kernel_wrapper_0(int k, int l,...)
{
    k = blockIdx.x*blockDim.x + threadIdx.x;
    l = blockIdx.y*blockDim.y + threadIdx.y;
    if (k >= 0 && k <= M - 1 && l >= 0 && l <= M - 1)
        kernel(k, l, ...);
}
```

But how to insert these guards?

The good news is that PIPS owns *preconditions* that are predicates on integer variables. Preconditions at entry of the kernel are:

```plaintext
// P(i, j, k, l) {0<=k, k<=63, 0<=l, l<=63}
```

Guard ≡ directly translation in C of preconditions on loop indices that are GPU thread indices.
Optimized reduction generation

- Reduction are common patterns that need special care to be correctly parallelized

\[ s = \sum_{i=0}^{N} x_i \]

- Reduction detection already implemented in PIPS
- Generate `#pragma omp reduce` in Par4All
- Generate GPU atomic operations
Communication optimization

- Naive approach: load/compute/store
- Useless communications if a data on GPU is not used on host between 2 kernels... 😞
- Use static interprocedural data-flow communications
  - Fuse various GPU arrays: remove GPU (de)allocation
  - Remove redundant communications
- \texttt{p4a --com-optimization} option since version 1.2
Loop fusion

- Programs ≡ often a succession of (parallel) loops
- Can be interesting to fuse loops together
  - Important for array-oriented languages: Fortran 95, Scilab, C++
  - parallel class...
  - Factorize control: one loop with bigger content
    - More important for heterogeneous accelerators: reduce kernel launch time
    - May avoid memory round trip
    - May cache recycling
- Use dependence graph, regions... to figure out when to fuse
- Sensible parallel promotion of scalar code to reduce parallelism interruption still to be implemented
Par4All Accel runtime

- CUDA or OpenCL can not be directly represented in the internal representation (IR, abstract syntax tree) such as `__device__` or `<<< >>>`
- PIPS motto: keep the IR as simple as possible by design
- Use some calls to intrinsics functions that can be represented directly
- Intrinsics functions are implemented with (macro-)functions
  - `p4a_accel.h` has indeed currently 2 implementations
    - `p4a_accel-CUDA.h` than can be compiled with CUDA for nVidia GPU execution
    - `p4a_accel-OpenCL.h` for OpenCL, written in C/CPP/C++
    - `p4a_accel-OpenMP.h` that can be compiled with an OpenMP compiler for simulation on a (multicore) CPU
- Add CUDA support for complex numbers
Par4All Accel runtime

- Can be used to simplify manual programming too (OpenCL...)
  - Manual radar electromagnetic simulation code @TB
  - One code targets CUDA/OpenCL/OpenMP
- OpenMP emulation for almost free
  - Use Valgrind to debug GPU-like and communication code! (Nice side effect of source-to-source...)
  - May even improve performance compared to native OpenMP generation because of memory layout change
Scilab language

- Interpreted scientific language widely used like Matlab
- Free software
- Roots in free version of Matlab from the 80’s
- Dynamic typing (scalars, vectors, (hyper)matrices, strings...)
- Many scientific functions, graphics...
- Double precision everywhere, even for loop indices (now)
- Slow because everything decided at runtime, garbage collecting
  - Implicit loops around each vector expression
    - Huge memory bandwidth used
    - Cache thrashing
    - Redundant control flow
- Strong commitment to develop Scilab through Scilab Enterprise, backed by a big user community, INRIA...
- HPC Project WildNode appliance with Scilab parallelization
- Reuse Par4All infrastructure to parallelize the code
Scilab & Matlab

- Scilab/Matlab input: *sequential* or array syntax
- Compilation to C code
- Parallelization of the generated C code
- Type inference to guess (crazy 😊) semantics
  - Heuristic: first encountered type is forever
- Speedup > 1000 😊
- Wild Cruncher: x86+GPU appliance with nice interface
  - Scilab — mathematical model & simulation
  - Par4All — automatic parallelization
  - //Geometry — polynomial-based 3D rendering & modelling
- Versions to compile to other platforms (fixed-point DSP...)

Par4All @ Paris HPC+GPU meetup

HPC Project — Wild Systems
R. Keryell
Wild Cruncher — Scilab parallelization

```scilab
// Compute the image
max = 200;
YI0 = 0.25;
YI1 = 0.8;
YI2 = 0.05;
Y0 = 0.85;
Y1 = 0.05;
Y2 = 0.45;
Type = "p";
X = 0.25;
Y = 0.625;
X1 = 0.25;
Y1 = 0.125;
X2 = 0.25;
Y2 = 0.9375;
while ((X+Y) < 4; k > max) 
  X = X + Y; 
  y = Y + X; 
  y = Y + X; 
  X = X; 
end;
```

R. Keryell

HPC Project — Wild Systems
• **Particle-Mesh** N-body cosmological simulation
• C code from Observatoire Astronomique de Strasbourg
• Use FFT 3D
• Example given in par4all.org distribution
void iteration(coord pos[NP][NP][NP],
    coord vel[NP][NP][NP],
    float dens[NP][NP][NP],
    int data[NP][NP][NP],
    int histo[NP][NP][NP]) {

    /* Split space into regular 3D grid: */
    discretisation(pos, data);

    /* Compute density on the grid: */
    histogram(data, histo);

    /* Compute attraction potential in Fourier’s space: */
    potential(histo, dens);

    /* Compute in each dimension the resulting forces and integrate the acceleration to update the speeds: */
    forcex(dens, force);
    updatevel(vel, force, data, 0, dt);
    forcey(dens, force);
    updatevel(vel, force, data, 1, dt);
    forcez(dens, force);
    updatevel(vel, force, data, 2, dt);

    /* Move the particles: */
    updatepos(pos, vel);
}

Par4All @ Paris HPC+GPU meetup
HPC Project — Wild Systems
R. Keryell
Stars-PM & Jacobi results

- 2 Xeon Nehalem X5670 (12 cores @ 2.93 GHz)
- 1 GPU nVidia Tesla C2050
- Automatic call to CuFFT instead of FFTW (stubs...)
- 150 iterations of Stars-PM

<table>
<thead>
<tr>
<th>Speed-up</th>
<th>p4a</th>
<th>Simulation Cosmo.</th>
<th>Jacobi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>32^3</td>
<td>64^3</td>
</tr>
<tr>
<td>Sequential (time in s)</td>
<td>(gcc -03)</td>
<td>0.68</td>
<td>6.30</td>
</tr>
<tr>
<td>OpenMP 6 threads</td>
<td>--openmp</td>
<td>4.25</td>
<td>4.92</td>
</tr>
<tr>
<td>CUDA base</td>
<td>--cuda</td>
<td>0.77</td>
<td>1.21</td>
</tr>
<tr>
<td>Optim. comm. 1.1</td>
<td>--cuda</td>
<td>3.4</td>
<td>5.38</td>
</tr>
<tr>
<td></td>
<td>--com-opt.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduction Optim. 1.1.2</td>
<td>--cuda</td>
<td>6.8</td>
<td>19.7</td>
</tr>
<tr>
<td></td>
<td>--com-opt.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manual optim.</td>
<td>(gcc -03)</td>
<td>13.6</td>
<td>24.2</td>
</tr>
</tbody>
</table>

p4a 1.1.2 introduce generation of CUDA atomic updates for PIPS detected reductions. Other solution to investigate: CuDPP call generation.
Benchmark results

With Par4All 1.2, CUDA 4.0, WildNode 2 Xeon Nehalem X5670 (12 cores @ 2.93 GHz) with nVidia C2050

From par4all.org distribution, in examples/Benchmarks
Hyantes

- Geographical application: library to compute neighbourhood population potential with scale control
- Example given in par4all.org distribution
- WildNode with 2 Intel Xeon X5670 @ 2.93GHz (12 cores) and an NVIDIA Tesla C2050 (Fermi), Linux/Ubuntu 10.04, gcc 4.4.3, CUDA 3.1
  - Sequential execution time on CPU: 30.355s
  - OpenMP parallel execution time on CPUs: 3.859s, speed-up: 7.87
  - CUDA parallel execution time on GPU: 0.441s, speed-up: 68.8
- With single precision on a HP EliteBook 8730w laptop (with an Intel Core2 Extreme Q9300 @ 2.53GHz (4 cores) and an NVIDIA GPU Quadro FX 3700M (16 multiprocessors, 128 cores, architecture 1.1)) with Linux/Debian/sid, gcc 4.4.5, CUDA 3.1:
  - Sequential execution time on CPU: 34.7s
  - OpenMP parallel execution time on CPUs: 13.7s, speed-up: 2.53
  - OpenMP emulation of GPU on CPUs: 9.7s, speed-up: 3.6
Results

Hyantes

CUDA parallel execution time on GPU: 1.57s, speed-up: 24.2

Original main C kernel:

```c
void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
        town pt[rangex][rangey], town t[nb])
{
    size_t i,j,k;

    fprintf(stderr,"begin_computation...
"");

    for(i=0;i<rangex;i++)
        for(j=0;j<rangey;j++) {
            pt[i][j].latitude = (xmin + step *i)*180/M_PI;
            pt[i][j].longitude = (ymin + step *j)*180/M_PI;
            pt[i][j].stock = 0.;

            for(k=0;k<nb;k++) {
                data_t tmp = 6368.* acos(cos(xmin + step *i)*cos(t[k].latitude) * cos((ymin + step *j) - t[k].longitude) + sin(xmin + step *i)*sin(t[k].latitude));
                if( tmp < range )
                    pt[i][j].stock += t[k].stock / (1 + tmp);
            }

            fprintf(stderr,"end_computation...
");
        }
}
```
OpenMP code:

```c
void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range, town pt[290][299]) {
    size_t i, j, k;

    fprintf(stderr, "begin_computation...
");  

#pragma omp parallel for private(k, j)
    for (i = 0; i <= 289; i += 1) 
        for (j = 0; j <= 298; j += 1) {
            pt[i][j].latitude = (xmin + step * i) * 180 / 3.14159265358979323846;
            pt[i][j].longitude = (ymin + step * j) * 180 / 3.14159265358979323846;
            pt[i][j].stock = 0.;
            for (k = 0; k <= 2877; k += 1) {
                data_t tmp = 6368. * acos(cos(xmin + step * i) * cos(t[k].latitude) * cos(ymin + step * j - t[k].longitude) + sin(xmin + step * i) * sin(t[k].latitude));
                if (tmp < range)
                    pt[i][j].stock += t[k].stock / (1 + tmp);
            }
    }

    fprintf(stderr, "end_computation...
");  
}

void display(town pt[290][299]) {

```
```c
size_t i, j;
for (i = 0; i <= 289; i += 1) {
    for (j = 0; j <= 298; j += 1)
        printf("%lf %lf %lf\n", pt[i][j].latitude, pt[i][j].longitude, pt[i][j].stock);
    printf("\n");
}
```
Generated GPU code:

```c
void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
          town pt[290][299], town t[2878])
{
    size_t i, j, k;
    //PIPS generated variable
    town (*P_0)[2878] = (town (*)[2878]) 0, (*P_1)[290][299] = (town (*)[290][299]) 0;

    fprintf(stderr, "begin_computation...
");
    P4A_accel_malloc(&P_1, sizeof(town[290][299])-1+1);
    P4A_accel_malloc(&P_0, sizeof(town[2878])-1+1);
    P4A_copy_to_accel(pt, *P_1, sizeof(town[290][299])-1+1);
    P4A_copy_to_accel(t, *P_0, sizeof(town[2878])-1+1);

    p4a_kernel_launcher_0(*P_1, range, step, *P_0, xmin, ymin);
    P4A_copy_from_accel(pt, *P_1, sizeof(town[290][299])-1+1);
    P4A_accel_free(*P_1);
    P4A_accel_free(*P_0);
    fprintf(stderr, "end_computation...
");
}

void p4a_kernel_launcher_0(town pt[290][299], data_t range, data_t step, town t[2878],
                           data_t xmin, data_t ymin)
{
    //PIPS generated variable
    size_t i, j, k;
    P4A_call_accel_kernel_2d(p4a_kernel_wrapper_0, 290,299, i, j, pt, range,
                              step, t, xmin, ymin);
}

P4A_accel_kernel_wrapper void p4a_kernel_wrapper_0(size_t i, size_t j, town pt[290][299],
```
data_t range, data_t step, town t[2878], data_t xmin, data_t ymin
{
    // Index has been replaced by P4A vp_0:
    i = P4A_vp_0;
    // Index has been replaced by P4A vp_1:
    j = P4A_vp_1;
    // Loop nest P4A end
    p4a_kernel_0(i, j, &pt[0][0], range, step, &t[0], xmin, ymin);
}

P4A_accel_kernel void p4a_kernel_0(size_t i, size_t j, town *pt, data_t range,
data_t step, town *t, data_t xmin, data_t ymin)
{
    //PIPS generated variable
    size_t k;
    // Loop nest P4A end
    if (i<=289&j<=298) {
        pt[299*i+j].latitude = (xmin+step*i)*180/3.14159265358979323846;
        pt[299*i+j].longitude = (ymin+step*j)*180/3.14159265358979323846;
        pt[299*i+j].stock = 0.;
        for(k = 0; k <= 2877; k += 1) {
            data_t tmp = 6368.*acos(cos(xmin+step*i)*cos((*(t+k)).latitude)*cos(ymin+step*j
            -(*(t+k)).longitude)+sin(xmin+step*i)*sin((*(t+k)).latitude));
            if (tmp<range)
                pt[299*i+j].stock += t[k].stock/(1+tmp);
        }
    }
}
Outline

1. HPC Project
2. Par4All
3. Scilab to OpenMP, CUDA & OpenCL
4. Results
5. Conclusion
6. Table des matières
Next event in the area

- Wild Cruncher UV
  - Scilab parallelization on SGI UV
  - 14/02/2012, SGI breakfast @ Novell France, Tour Franklin, La Défense
Conclusion

- GPU (and other heterogeneous accelerators): impressive peak performances and memory bandwidth, power efficient
- Domain is maturing: any languages, libraries, applications, tools... Just choose the good one 😊
- Real codes are often not well written to be parallelized... even by human being 😐
- At least writing clean C99/Fortran/Scilab... code should be a prerequisite
- Take a positive attitude... Parallelization is a good opportunity for deep cleaning (refactoring, modernization...) → improve also the original code
- Open standards to avoid sticking to some architectures
- Need software tools and environments that will last through business plans or companies
Conclusion

- Open implementations are a warranty for long time support for a technology (cf. current tendency in military and national security projects)
- p4a motto: keep things simple
- Open Source for community network effect
- Easy way to begin with parallel programming
- Source-to-source
  - Give some programming examples
  - Good start that can be reworked upon
- Entry cost
- Exit cost! 😊
  - Do not loose control on *your* code and *your* data!
- HPC Project is hiring 😊
Outline

1. HPC Project
2. Par4All
3. Scilab to OpenMP, CUDA & OpenCL
4. Results
5. Conclusion
6. Table des matières