Perspectives in parallel programming

MARCO DANELUTTO
PHD COURSE
MAY 2015
Algorithmic skeletons

Origin from the HPC community

◦ Murray Cole thesis

Concept

Here, a higher order function could be represented as a program or procedure “template”, specifying the overall structure of a computation, with gaps left for the definitions of problem specific procedures and declarations. The fit would be somewhat better in languages which allow procedures to be passed as parameters. Thus, the system which we are about to propose could be presented to the programmer in the context of any “base” language, whether declarative or imperative. Our subsequent use of the term “higher order function” should be interpreted with this freedom in mind. The programming model which we intend to investigate can now be described. In this model, the programmer is presented with a selection of specialised higher order functions (or similar, depending upon the base language) from which one must be selected as the outermost function in the program.
The principle

- The new system presents the user with a selection of independent “algorithmic skeleton”, each of which describes the structure of a particular style of algorithm, in the way in which “higher order functions” represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.

(Cole 1988)
The principle (rephrased)

• Abstract parallelism exploitation pattern by parametric code (higher order function)

• Provide user mechanism to specify the parameters (sequential code, extra parameters)

• Provide (user protected) state-of-the-art implementation of each parallelism exploitation pattern

• In case, allow composition

  • Fundamental, property not present in first skeletons systems
Sample pattern: the task farm

- **Parameters:**
  - Worker code
  - Parallelism degree (computed?)

- **Known implementation**
  - Master slave pattern
  - Possibly distributed master

- **Composite worker**
  - Master to master optimizations
More skeletons

Pipeline
Farm
Map
Reduce
Scan
Stencil
Divide&conquer
Pipeline

Computation in stages

Variants:
- Parallel stages
- Stateful stages
- Feedback

\[ Ts = \max \{ T_{si} \} \]

\[ T_c \approx m \times Ts \]
Embarrassingly parallel computations on streams

\[ Ts = \frac{Tw}{nw} \]

\[ Tc \approx m \times Ts \]
Map

Embarrassingly parallel computations on data collection

$L = \frac{Tw}{nw}$
Reduce

Summing up a collection

Associative and commutative operator

$L = \log(N) \times Tw$
Scan (parallel prefix)

Summing up a collection

Associative and commutative operator

Keep intermediate results

\[ L = N \times \log(N) \times Tw \]
Cole’s manifesto

1. Propagate the concept with minimal disruption
   - No chance to introduce yet another parallel programming language

2. Integrate ad hoc parallelism
   - Specialized, ad hoc solutions must be hosted

3. Accommodate diversity
   - Slightly different skeletons should be derivable

4. Show the payback
   - Advertising: demonstrate that moving to skeletons is worthwhile
Our additions (PARCO 2005)

5 Support code reuse
   - Huge amount of (dusty deck?) code
   - Large amounts of (open source) “libraries”

6 Handle heterogeneity
   - Cluster/networks/grids are heterogeneous
   - Upgrades of clusters (with different release procs and different amounts/speed of main store)

7 Handle dynamicity
   - Non dedicated computing nodes (varying load)
   - Different nodes, different power
Cole PhD Thesis

• Fixed degree D&C, Iterative Combination (2 “best” items in the set combined, iterated), Cluster Skeleton (abstract machine rather than algorithm), Task Queue

• Lot of usage examples and analytical evaluation of skeletons

• Seminal work in the area
  • Due to the motivations
  • More that to the skeletons discussed

• Hierarchical composition later on (‘95 PARCO)
Evolution of the concept

- Initial frameworks
- Homogeneous hw targeting
- Heterogeneous hw targeting
Skeleton evolution

Cole PhD (1988)
Fixed degree DC, Iterative combination, Cluster Task queue

Darlington (1992)
Pipeline, Farm, RaMP, DMPA
SCL Fortran S

P3L (1991)
Pipeline, Farm, Map, Reduce
SkIE ASSIST Lithium OcamlP3L

BMF ('80)
map fold reduce prefix + algebra

MALLBA ('00)
Combinatorial optimisation

Skillicorn (mid '90)

Muesli (2002)
Pipeline, Farm, Parallel array + collectives

Kuchen Skil (1998)

Serot (1999)
Skipper (→ MDF)

Gorlatch (late '90)
HOC (early '00)

eSkel (2002)
Parametric skeletons + Give/Take
Cooperation comes in

This is in contrast to the low-level parallel extensions to languages where both tasks must be programmed simultaneously in an unstructured way. The coordination approach provides a promising way to achieve the following important goals:

- **Reusability of Sequential Code**: Parallel programs can be developed by using the coordination language to compose existing modules written in conventional languages.

- **Generality and Heterogeneity**: Coordination languages are independent of any base computational language. Thus, they can be used to compose sequential programs written in any language and can, in principle, coordinate programs written in several different languages.

- **Portability**: Parallel programs can be efficiently implemented on a wide range of parallel machines by specialised implementations of the compositional operators for target architectures.

Darlington et al. Functional Skeletons for Parallel Coordination (Europar '95)
Darlington (2)

- **Initially (‘91)**
  Farm, Pipeline, RaMP, DMP

- **Then (‘95):**

  - Coordination (see before)
  - Clearer data parallel asset
  - Control parallel skeletons (Farm, SPMD)
  - Transformations!
  - Fortran embedding!
Kuchen: Muesli

- Clearly separates data and control parallelism exploitat...

- Builds on top of MPI

- Inherits two tier model from P3L:
  - Arbitrary control parallel nestings
  - With data parallel or sequential leaves
int main(int argc, char **argv){
    try{
        InitSkeletons(argc,argv);

        Initial<int> p1(init);
        Atomic<int,int> p2(square,1);
        Process* p3 = NestedFarm<int,int>(p2,4);
        Final<int> p4(fin);
        Pipe p5(p1,*p3,p4);

        p5.start();

        TerminateSkeletons();}
    catch(Exception&){cout << "Exception" << endl << flush;}
}
Muesli : data parallel

template <class C> // using algorithm of Gentleman based on torus topology
DistributedMatrix<C> matmult(DistributedMatrix<C> A, DistributedMatrix<C> B) {
    A.rotateRows(& negate);
    B.rotateCols(& negate);
    DistributedMatrix<C> R(A.getRows(), A.getCols(), 0,
                            A.getBlocksInCol(), A.getBlocksInRow());
    for (int i = 0; i < A.getBlocksInRow(); ++i) {
        typedef C (*skprod_t)(const DistributedMatrix<C>&,
                               const DistributedMatrix<C>&, int, int, C);
        R.mapIndexInPlace(cryo((skprod_t)skprod<C>)(A)(B));
        A.rotateRows(-1);
        B.rotateCols(-1);
    }
    return R;
}

int main(int argc, char **argv) {
    try {
        InitSkeletons(argc, argv);
        DistributedMatrix<int> A(Problemsize, Problemsize, & add, sqrt, sqrt);
        DistributedMatrix<int> B(Problemsize, Problemsize, & add, sqrt, sqrt);
        DistributedMatrix<int> C = matmult(A, B);
        TerminateSkeletons();
        catch (Exception & e) { cout << "Exception" << endl << flush; }
    }
}
Cole eSkel

- Local data or Spread data processing
- Implicit or explicit interaction mode
- Transient and persistent skeleton calls in a skeleton
- Pipeline, Deal (cyclic distrib farm), Farm, Butterfly
- MPI (rather glossy interface)
MALLBA

- Combinatorial optimization through skeletons
- Fairly “unconventional” set of skeletons
  - D&C, B&B, Dynamic Programming, Hill Climbing, Metropolis, Simulated Annealing (SA), Tabu Search (TS) and Genetic Algorithms (GA)
- C++ implementation
  - provided classes (fixed implementation) + required classes (user supplied, problem dependent code)
- Related work on performance models
- Excellent speedups on (heterogeneous CPU) clusters as well as on WAN
Gorlatch: HOC

- Inherits from Lithium
- Exploiting Web Services
- Higher order components
  - Farms, pipelines
- Developed in Muenster
- Joint works with
  - Caramel, Cole, Danelutto

```java
public interface Worker {
    public double[] compute(double[] input);
}
public interface Master {
    public double[][] split (double[] input, int numWorkers);
    public double[] join(double[][] input);
}

farmHOC = farmFactory.createHOC();
farmHOC.setMaster("masterID"); // web service invocation in Java
farmHOC.setWorker("workerID");
String[] targetHosts = {"masterH", "workerH1", ...};
farmHOC.configureGrid(targetHosts); // deployment of the farmHOC on the Grid
farmHOC.compute(input);
```
The Pisa picture

P3L (the Pisa Parallel Programming Language 1991)

SkIE
(Skeleton Integrated Environment 1997)

OcamLP3L
(1998)

Muesli
(two tier model)

Macro Data Flow RunTime (1999)

SKElab (SKEleton LIBrary 2000)

eSkel
(library concept)

Skipper
(MDF impl)

Lithium (2000)

HOC
(interpreter)

muskel
(μskeleton lib 2003)

ASSIST
(A Software development System based on Integrated Skeleton Technology 2001)

Distributed systems: paradigms and models (M. Danelutto)
P3L
C+MPI
Anacleto
COW
HP Pisa
ScienceCenter
1990

seq pixel in (float init[2], float end[2], int k, int resolution)
  out(int color[400])
end seq

farm mandelbrot in (float init[2], float end[2], int k, int resolution)
  out(int color [400])
  pixel in(init, end, k, resolution) out(color)
end farm

pipe main in() out()
  start in() out(float init[2], float end[2], int k, int res)
  mandelbrot in(init, end, k, res) out(int color[400])
  stop in(color) out()
end pipe
**SKElib: Parallel Programming with Skeletons in C**

Marco Danelutto and Massimiliano Stigliani

Department Computer Science – University of Pisa
Corso Italia, 40 – I-56125 Pisa – Italy
marcod@di.unipi.it, massimiliano.stigliani

```c
#include "ske.h"

extern void f(P_IN * in, P_OUT * out);
extern void g(P_OUT * in, G_OUT * out);

int main(int argc, char *argv[]) {
    SKELETON * seqf, * seqg, * farmf, * the_pipe;
    int n_workers = atoi(argv[1]);
    ...
    seqf = SKE_SEQ((FUN *)f, sizeof(P_IN), sizeof(P_OUT));
    seqg = SKE_SEQ((FUN *)g, sizeof(P_OUT), sizeof(G_OUT));
    farmf = SKE_FARM(n_workers, seqf, BALANCING);
    the_pipe = SKE_PIPE(2, farmf, seqg);
    ...
    n_hosts = atoi(argv[2]);
    SKE_CALL(the_pipe, OPTIMIZE, "input.dat", "output.dat",
             n_hosts, "alpha1", "alpha2", "alpha3", "alpha4", "alpha5");
    ...
    return(0);
}
```

---

**EuroPar 2000**
OcamłP3L (1996 + 2003-2007)

```ocaml
1. let solver =
2.     parfun (fun () ->
3.       (loop ((fun (v,continue) -> continue ),
4.         seq(fun _ -> fun (v,_) -> v)
5.         ||| mapvector(seq(fun _ -> calcul_sous_domaine),3)
6.         ||| seq(fun _ -> projection)
7.         ||| seq(fun _ -> bicgstab)
8.         ||| seq(fun _ -> plot)
9.         ) ) ) ;;

10. pardo( fun () ->
11.     List.iter print_result
12.     (P31stream.to_list (solver (P31stream.of_fun generate_input_stream)))
13. );;
```

Figure 4: Code fragment from a Poisson solver.
Introducing the Muskel (2005-2008) manager concept
Initial skeleton frameworks

Different set of skeletons
- Map, reduce, scan, pipeline, ...

Different *host* framework
- Fortran S, C, Java

Different *target* framework
- Processes, Sys V IPC, TPC/IP sockets

Different implementations
- Languages, libraries
Evolution of the concept

- Initial frameworks
- Homogeneous hw targeting
- Heterogeneous hw targeting
Homogeneous hw targeting

Cluster of workstations
- COW, NOW, GRIDs (, clouds ...)

Multiprocessors
- SMP, multicore, many core (GP)
More skeleton frameworks

SKETO (2003 - )
Library
Minimal disruption
Data parallel (map fusion)

Muesli (2005-)
Target COW (MPI) & smp (OpenMP)
C++ integration (functional)

Macro data flow
ocaml

OSL (2008-)
Bsp style
COQ integration

FastFlow (2008 - )
Accelerator concept
Fast comm
C++ hdr only
Achievements

Data & Stream parallelism
  ◦ Both needed although in different phases

Nesting
  ◦ Free nesting of skeletons
  ◦ Composition model (single entry/exit point)
  ◦ Two tier model (P3L, Kuchen)
    ◦ Stream parallel, then
    ◦ Data parallel, then
    ◦ Sequential computations
Achievements

Library (vs new languages)
- Seamless integration
- Dynamic optimization
- Advanced language features exploited
  - Templates in C++
Achievements

Need for library/compiler supported optimizations

- Hardware targeting
  - Through different library entries
- Parallelism degree
  - Automatically devised
- Composition
  - Automatic
- Mapping/scheduling
  - Automatic
Parallelism degree

Bottom up
- Profile seq code
- Use models to figure out *optimal* pardegree

Top down
- Reduce resource
- According to the models
- Up to the resource availability is matched

(P3L)
Achievements

Rewriting rules
- Formal rewriting rules
- Preserve functional semantics
- Alter non functional semantics
- Sample: \( \text{comp}(\text{map}(f), \text{map}(g)) = \text{map}(\text{comp}(f, g)) \)

Rewrite based optimization
- Re-writing of the skeleton tree
- Performance model driven
Normal form

For stream parallel skeleton tree
- Visit fringe (left to right)
- Create a COMP (sequential composition)
- Use if as a worker in a farm

Guarantees optimization of
- Communication, synchronization
- Only for throughput
Evolution of the concept

- Initial frameworks
- Homogeneous hw targeting
- Heterogeneous hw targeting
# SkeCL

```cpp
#include <SkeCl/SkeCL.h>
#include <SkeCl/zip.h>
#include <SkeCl/Reduce.h>
#include <SkeCl/Vector.h>

using namespace skecl;

int main() {
  skecl::init(); // initialize SkeCL

  // specify calculations using parallel patterns (skeletions):
  Zip<int, int> mult("int func(int x, int y) { return x*y; }");
  Reduce<int> sum("int func(int x, int y) { return x+y; }", 0);

  Vector<int> A(1024); Vector<int, B(1024); // create and fill vectors
  init(A.begin(), A.end()); init(B.begin(), B.end());

  Vector<int> C = sum(mult(A, B)); // perform calculation in parallel

  std::cout << "Dot product: " << C.front() << std::endl; // access result
}
```

---

**Marrow Algorithmic Skeleton Library**

Marrow is a C++ framework for the construction and execution of complex computations in multi-GPU GPUs (OpenCL enabled devices) while addressing modularity and performance concerns. Modularity is achieved by using skeleton nesting, a novel approach in the GPU-GPU skeletons context, which allows the composition of basic constructs to create complex and structured computations. This feature enables a higher degree of control from Marrow, by abstracting individual skeleton execution from the programmer and enabling the library to integrate optimizations transparently.

---

**Algorithmic skeletons for multi-core, multi-GPU systems and clusters**

**Steffen Ernsting* and Herbert Kuchen**

Department of Information Systems, University of Muenster, Leonardo-Campus 3, 48149 Muenster, Germany
E-mail: steffen.ernsting@wi.uni-muenster.de
E-mail: kuchen@wi.uni-muenster.de

*Corresponding author
Targeting
#include "skepu/vector.h"
#include "skepu/mapreduce.h"
#include "skepu/map.h"
#include "skepu/reduce.h"

// User-function used for mapping
BINARY_FUNC(mult_f, float, a, b,
    return a*b;
)

// User-function used for reduction
BINARY_FUNC(plus_f, float, a, b,
    return a+b;
)

#define N (1024*1024)
//}}def N

int main()
{
    skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f, new plus_f);
    skepu::Map<mult_f> mapzip(new mult_f);
    skepu::Reduce<plus_f> redsum(new plus_f);
    skepu::Vector<float> v0(N, (float)2);
    skepu::Vector<float> v1(N, (float)5);
    skepu::Vector<float> vtemp(N, (float)0);

    std::cout << "Computing mapreduce version " << std::endl;
    #ifdef SKEPU_OPENMP
    clock_t t0, t1;
    t0 = clock();
    #endif
    #ifdef SKEPU_CUDA
    cudaEvent_t t0, t1;
    cudaEventCreate(&t0);
    cudaEventCreate(&t1);
    cudaEventRecord(t0, 0);
    #endif
    float r = dotProduct(v0, v1);
    #ifdef SKEPU_OPENMP
    t1 = clock();
    float elapsed;
    elapsed = ((float) (t1-t0)) / 1000.0;
    elapsed = elapsed / CLOCKS_PER_SEC;
    std::cout << "Elapsed time (OpenMP) " << elapsed << " msecs" << std::endl;
    #endif
    #ifdef SKEPU_CUDA
    float elapsed;
    cudaEventRecord(t1, 0);
    cudaEventSynchronize(t1);
    cudaEventElapsedTime(&elapsed, t0, t1);
    std::cout << "Elapsed time (CUDA) " << elapsed << " msecs" << std::endl;
    #endif
}
ABSTRACT

SkePU is a C++ template library that provides a simple and unified interface for specifying data-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL. The interface is also general enough to support other architectures, and SkePU implements both a sequential CPU and a parallel OpenMP backend. It also supports multi-GPU systems. Currently available skeletons in SkePU include map, reduce, mapreduce, map-with-overlap, map-array, and scan. The performance of SkePU generated code is comparable to that of hand-written code, even for more complex applications such as ODE solving.

In this paper, we discuss initial results from auto-tuning SkePU using an off-line, machine learning approach where we adapt skeletons to a given platform using training data. The prediction mechanism at execution time uses off-line pre-calculated estimates to construct an execution plan for any desired configuration with minimal overhead. The prediction mechanism accurately predicts execution time for repetitive executions and includes a mechanism to predict execution time for user functions of different complexity. The tuning framework covers selection between different backends as well as choosing optimal parameter values for the selected backend. We will discuss our approach and initial results obtained for different skeletons (map, mapreduce, reduce).
Autonomic scheduling of tasks from data parallel patterns to CPU/GPU core mixes

T. Serban, M. Danelutto
Dept. Computer Science
Univ. of Pisa – Italy

P. Kilpatrick
Dept. Computer Science
Queen’s Univ. of Belfast – UK

Abstract—We propose a methodology for optimizing the execution of data parallel (sub-)tasks on CPU and GPU cores of the same heterogeneous architecture. The methodology is based on two main components: i) an analytical performance model for scheduling tasks among CPU and GPU cores, such that the global execution time of the overall data parallel pattern is optimized; and ii) an autonomic module which uses the analytical performance model to implement the data parallel computations in a completely autonomic way, requiring no programmer intervention to optimize the computation across CPU and GPU cores.

The analytical performance model uses a small set of simple parameters to devise a partitioning—between CPU and GPU cores—of the tasks derived from structured data parallel patterns/algorithmic skeletons. The model takes into account both hardware related and application dependent parameters. It computes the percentage of tasks to be executed on CPU and GPU cores such that both kinds of cores are exploited and performance figures are optimized.

The autonomic module, implemented in FastFlow, executes a generic map (reduce) data parallel pattern scheduling part of the tasks to the GPU and part to CPU cores so as to achieve optimal execution time.

Experimental results on state-of-the-art CPU/GPU architectures are shown that assess both performance model properties and autonomic module effectiveness.

\[
P = \frac{2 \times T_g}{D} + \frac{2 \times S}{T_r} + \frac{T_{f2}}{G} \frac{1}{G} + \frac{2 \times S}{T_r} + \frac{T_{f2}}{G}
\]
Abstract—The single core processor, which has dominated for over 30 years, is now obsolete with recent trends increasing towards parallel systems, demanding a huge shift in programming techniques and practices. Moreover, we are rapidly moving towards an age where almost all programming will be targeting parallel systems. Parallel hardware is rapidly evolving, with large heterogeneous systems, typically comprising a mixture of CPUs and GPUs, becoming the mainstream. Additionally, with this increasing heterogeneity comes increasing complexity: not only does the programmer have to worry about where and how to express the parallelism, they must also express an efficient mapping of resources to the available system. This generally requires in-depth expert knowledge that most application programmers do not have. In this paper we describe a new technique that derives, automatically, optimal mappings for an application onto a heterogeneous architecture, using a Monte Carlo Tree Search algorithm. Our technique exploits high-level design patterns, targeting a set of well-specified parallel skeletons. We demonstrate that our MCTS on a convolution example obtained on average 95% of the speed up achieved by the hand tune optimisation developed by expert user.
FPGAs
More coming

Figure 2.1: Overview of the Workflow using ThreadpoolComposer