Utilizing Modern Programming Techniques and the Boost Libraries for Scientific Software Development

Josef Weinbub, Karl Rupp, and Siegfried Selberherr

Institute for Microelectronics
Technische Universität Wien
Vienna - Austria - Europe
http://www.iue.tuwien.ac.at

What is it About?

Generic/Functional/Meta-programming

Boost Libraries: Graph, Fusion, MPL, Phoenix, ...

Three application scenarios:

- Task Scheduler
- Meta-Property Selection
- Algorithm Generalization

The Setting

Boost provides a vast set of functionality for free

But - basic C++ programmers might get deterred by ..

.. advanced techniques: concepts, traits, meta-functions, ..

The Setting

Boost Libraries and modern programming techniques

More and more utilized in scientific/engineering implementations

Gaining additional skills pays off in terms of productivity

→ It does make sense to go for advanced C++ skills!

Let's Get Started!

© *Gertfrik* | *StockFreeImages.com*

Schedule Tasks

Extendable component/plugin/task framework

Use tasks to setup an intricate execution chain

Tasks have dependencies

Schedule Tasks

Common Approach: Task Graph

Map tasks to vertices

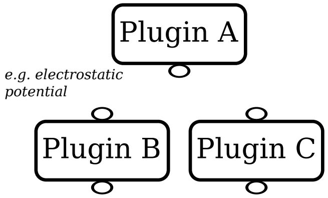
Map task dependencies to edges

Schedule Tasks

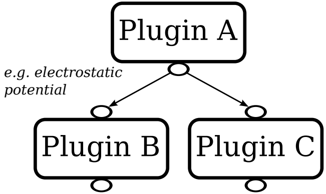
Plugin A

e.g. electrostatic potential

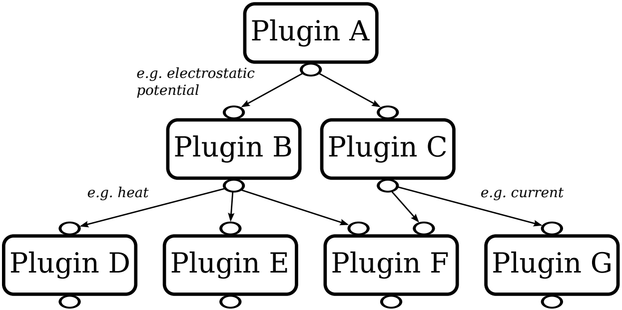
Schedule Tasks



Schedule Tasks



Schedule Tasks



Schedule Tasks

Utilize Boost.Graph!

Mature implementation: since 2000

Flexible graph datastructures: (un)directed, etc..

Many algorithms available: BFS, DFS, etc..

Graph Definition

```
typedef boost::adjacency_list<
   boost::vecS,       // vertex container type
   boost::vecS,       // edge container type
   boost::directedS,  // graph type
   boost::property<boost::vertex_name_t,
    std::string>
> Graph;
Graph graph;
```

Note the generic graph setup: non-intrusive datastructure definition

Add Vertices

C++11

Range-based for-loops!

```
for(plugin* pl : plugins) {
  boost::add_vertex(pl->name(), graph);
}
```

Add Edges

C++11

Range-based for-loops!

```
for(plugin* pl : plugins) {
  for(input in : pl->input()) {
    /* find the vertex/plugin which provides the
       required input */
    boost::add_edge(source_id, sink_id, graph);
}}
```

Let's Schedule: Sequential Execution

Based on available task graph

Utilize list scheduling approach

List Scheduling

Setup a list of prioritized tasks, and process them repeatedly until all tasks are dealt with.

Let's Schedule: Sequential Execution

Essential step: prioritized tasks

based on dependencies

→ Topological Sort

Prioritize

```
typedef std::list<Vertex>   PriorList;
PriorList prioritized;

boost::topological_sort(graph,
  std::front_inserter(prioritized));
```

1.	2.	3.	4.	5.	6.	7.
Plugin A	Plugin B	Plugin C	Plugin D	Plugin E	Plugin F	Plugin G

STL Style Processing

```
for(Vertex v : prioritized){
  if(is_executable(v)) execute(v);
}
```

Phoenix Style Processing

```
std::for_each(prioritized.begin(), prioritized.end(),
  if_(is_executable) [ execute ] );
```

Why a Boost.Phoenix Implementation?

Intuitive, Concise

In-place functional expressions

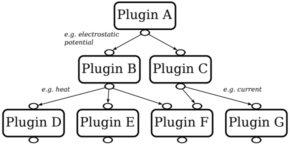
→ Increased information density

Let's Parallelize

Sequential approach directly parallelizable

Only real change: task execution implementation

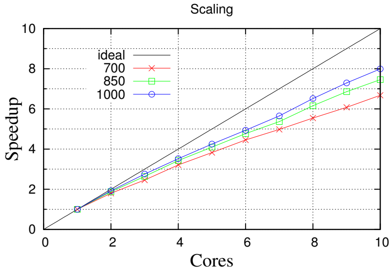
→ distribute via MPI

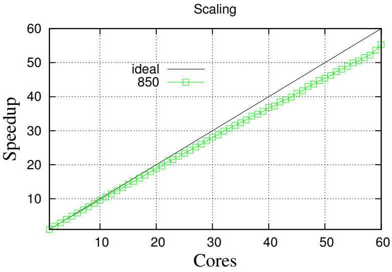


Results

Dense matrix matrix product

Different problem sizes





In Conclusion

Boost.Graph does 80% of the work

30-50 lines of code: graph and prioritization

Sequential and parallel implementation difference: task execution

Functional traversal/execution: intuitive, concise

Onward!

©*Matt Wedel* | *http://svpow.com/*

Meta-Selection

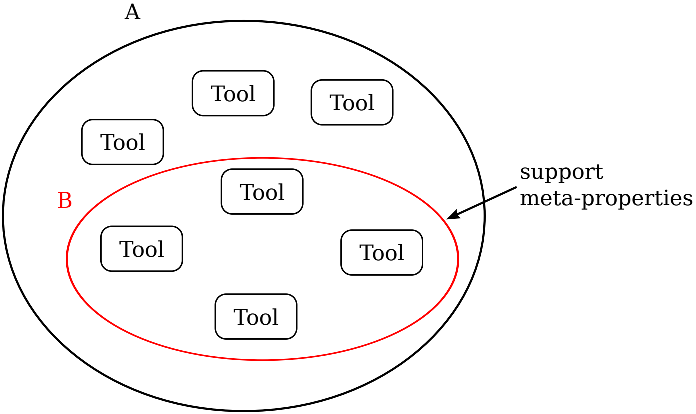
Compile-time component selection

Set of components

Attach properties (non)intrusively

Select components based on set of properties

Meta-Selection



Applications

Mesh generation/adaptation tools

- Dimensionality
- Mesh element
- Algorithm

Algorithms

- Dimensionality
- (Non)Robust
- Coordinate system (Boost.Geometry)

Meta-Selection

Utilize Boost.Fusion Library

Utilize Boost.Metaprogramming Library

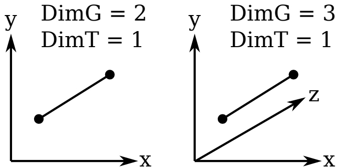
Utilize Boost.TypeTraits Library

Extends: `filter_view` algorithm

Attach Meta-Properties to Components: Intrusively

Intrusive approach

```
struct mesh_generator_one {
 // ...
 typedef result_of::make_map<
  dimg, dimt, cell,
  three,three,simplex>::type  properties_type;};
```



Attach Meta-Properties to Components: Non-Intrusively

Non-intrusive approach

```
namespace result_of {
template<typename T>
struct properties{typedef error type;};

template<>
struct properties <mesh_generator_one> {
 typedef typename result_of::make_map<
   dimg, dimt, cell,
   three,three,simplex,
 >::type    type; };
}
```

Determine the Subset

User-level code

```
typedef vector<Tool1,Tool2..>      AvailableTools;

typedef result_of::make_map<
   dimg, dimt, cell,
   three,three,simplex,
>::type                            Properties;

typedef typename filter_fold::apply<
  Properties, AvailableTools
  >::type                          ResultTools;
```

Internals: filter_fold

```
struct filter_fold {
  struct fold_op {
    template <typename Sig>    struct result;

    template <class S, class ToolSet, class Property>
    struct result< S(ToolSet &,Property &) > {
      typedef typename mpl::filter_view<
        ToolSet, check<Property>
      >::type  type; }; };

  template <typename Properties, typename ToolSet>
  struct apply : fusion::result_of::fold<
    Properties, ToolSet, fold_op>::type { }; };
```

More Internals: check<Property>

```
template<typename PairT>
struct check {
  template<typename EleT>
  struct apply {
    typedef typename result_of::value_of<
      typename result_of::find_if<
        typename result_of::properties<EleT>::type,
        is_same<_,PairT>
      >::type
    >::type                          find_result_type;
  // process to true/false type-member
```

In Conclusion

50 lines of code: meta-selection facility

Boost does the majority of the work

Highly extendible, flexible, and non-intrusive approach

Arbitrary number and types of properties

Keep Going!

©*http://morguefile.com/*

Algorithm Generalization

Apply the generic paradigm

Not so much about the Boost libraries

Field of application: Computational Geometry

Lift geometric algorithm interfaces

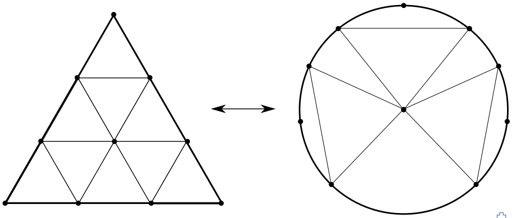
Geometry / Topology

Geometry

Deals with shape, size, position

Topology

Continuity, connectivity



Use Geometry and Topology

Geometrical algorithm contains

- Geometrical information, ie., geometrical space R^d
- Topological information, ie., number of vertices of the underlying polygon

```
area_triangle(Vector p1, Vector p2, Vector p3);
```

Note

Boost.Geometry generalizes: polygon

Algorithm Investigations

	Geometry	Topology
Line Length, $\mathbb{R}^3$	3D	1D,`S`/`C`
Triangle Area, $\mathbb{R}^2$	2D	2D,`S`
Tetrahedron Volume, $\mathbb{R}^3$	3D	3D,`S`
Cube Volume, $\mathbb{R}^3$	3D	3D,`C`

Topology

`S`,`C` denotes simplex and cube topology

k-Cell to the Rescue!

Topology

Map geometrical entity to a k-cell

k-cell	topological object	geometrical object
0-cell	Vertex	Point
1-cell	Edge	Line
2-cell	Face	Triangle, Quadrilateral, ...
3-cell	Cell	Tetrahedron, Cuboid, ...

Topology

Well-defined mapping only for $k \leq 1$

Mapping

Topology

Well-defined for $k > 1$ only with topology

k-cell	Cell Topology	Geometrical Entity
0-cell	Simplex/Cube	Point
1-cell	Simplex/Cube	Line
2-cell	Simplex	Triangle
2-cell	Cube	Quadrilateral
3-cell	Simplex	Tetrahedron
3-cell	Cube	Cuboid

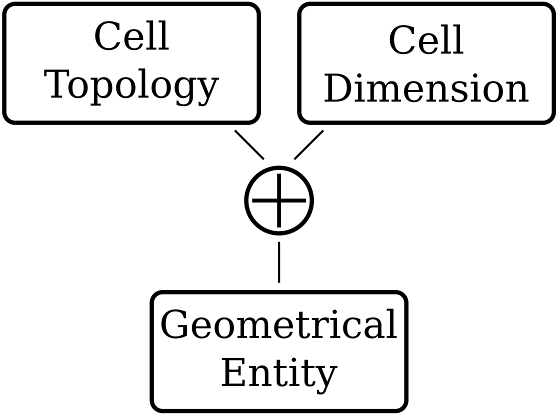
Important!

Abstract a geometrical by a topological entity

Topology dimension

Cell topology

Abstraction



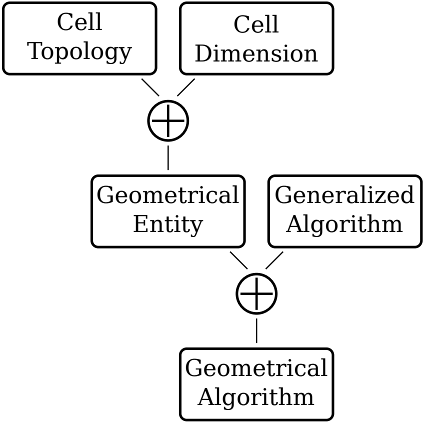
Abstract Geometrical Algorithms

algorithm	**generalized algorithm**
Length of a line	Metric quantity
Area of a triangle	Metric quantity
Volume of a tetrahedron	Metric quantity
Point in triangle test	k-cell in q-cell
Point in tetrahedron test	k-cell in q-cell

- No dimensionality
- No indication of a geometrical entity

$\rightarrow$ Only the essence! $\rightarrow$ Reflects the generic paradigm!

Abstraction



Aren't We at a Programming Conference?

©*Starush* | *StockFreeImages.com*

Map to Implementation

Use partial template specialization

```
template < int Dimension , typename Topology >
struct metric_quantity_impl { };
```

User level code

```
typedef boost::result_of<
  metric_quantity(Cell) >::type quan_type;
quan_type quan = metric_quantity()(*cit);
```

Specialization: 1D Distance

```
template < typename Topology >
struct metric_quantity_impl < 1, Topology > {
  template<class> struct result;

  template<class F, typename Cell>
  struct result<F(Cell)> {
    // cell dependent return-type
    typedef double type;
  };

  template < typename Cell >
  typename result< metric_quantity_impl(Cell) >::type
  operator()(Cell& cell) const {
    return boost::geometry::distance(cell[0],cell[1]);
  }};
```

Specialization: 1D Distance - Different Algorithm

```
template < typename Topology >
struct metric_quantity_impl < 1, Topology > {
  template<class> struct result;

  template<class F, typename Cell>
  struct result<F(Cell)> {
    // cell dependent return-type
    typedef double type;
  };

  template < typename Cell >
  typename result< metric_quantity_impl(Cell) >::type
  operator()(Cell& cell) const {
    return my_line_distance(cell[0],cell[1]);
  }};
```

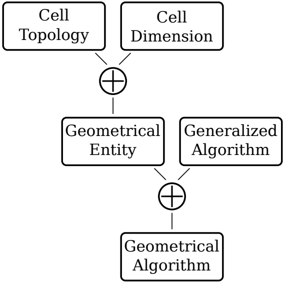
Specialization: 2D Simplex Area

```
template < >
struct metric_quantity_impl < 2, tag::simplex > {
  template<class> struct result;

  template<class F, typename Cell>
  struct result<F(Cell)> {
    // use a high-precision floating-point datatype,
    // e.g., ARPREC
    typedef mp_real type;
  };

  template < typename Cell >
  typename result< metric_quantity_impl(Cell) >::type
  operator()(Cell& cell) const {
    return boost::geometry::area(cell);
  }};
```

Analysis



Generalized Algorithm

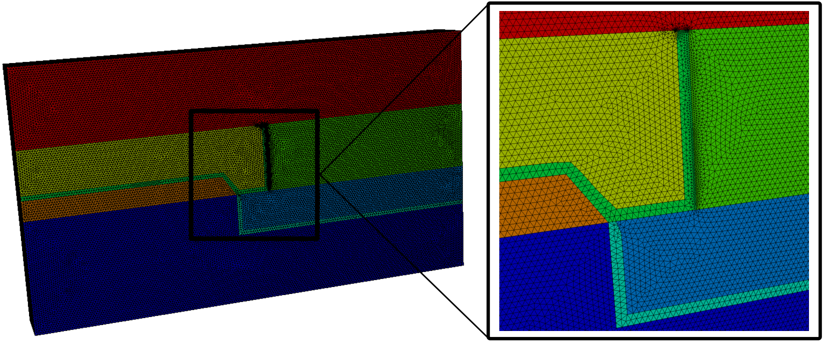
```
template<int Dimension,typename Topology>
struct metric_quantity{};
```

Geometrical Algorithm

```
template<>
struct metric_quantity<2,simplex>{..};
```

Cell Dimension Cell Topology

Application: Mesh-Based Algorithms



Et voilà: Meshtype-Independent Algorithms!

```
typedef config::triangular_2d                Config;
typedef result_of::domain<Config>::type      Domain;
Domain domain;
// fill the mesh domain

typedef boost::result_of<
  metric_quantity(Cell) >::type quan_type;
..
CellRange cells = ncells(domain);
for(CellIterator cit = cells.begin();
    cit != cells.end();++cit) {
 quan_type quan = metric_quantity()(*cit);
}
```

ViennaGrid http://viennagrid.sourceforge.net/

In Conclusion

Highly generalized/abstracted implementations

Extendible interface based on basic technique

Theoretical generalization approach directly implementable

Works best with a compile-time mesh datastructure

We Did It!

©Jborzicchi | StockFreeImages.com

What Did We Talk About?

Different application cases have been introduced

- Task scheduler
- Meta-property selection
- Algorithm generalization

Utilized generic/function/meta programming techniques

Utilized the Boost libraries

What Did We Talk About?

Highly versatile, maintainable, and extendible code

Actual implementation effort kept to a minimum

Boost libraries do the majority of the work

→ It is worth the effort!