Zalán Szűgyi, Norbert Pataki

A More Efficient and Type-Safe Version of FastFlow

Eötvös Loránd University, Faculty of Informatics

the Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003)

WGT 2011

Outline

- Introduction to FastFlow
- Type-safety problems of FastFlow
- Solution proposal, CRTP
- Implementation details
  - Buffers
  - Farm
  - Pipeline
- Example, and conclusion

FastFlow

- Parallel programming framework for multi-core platforms
- Based on non-blocking lock-free / fence-free synchronization
- Composed stack of layers:
  - ease of programming
  - fast and scalable
- Targeted to streaming application

Layers of FastFlow

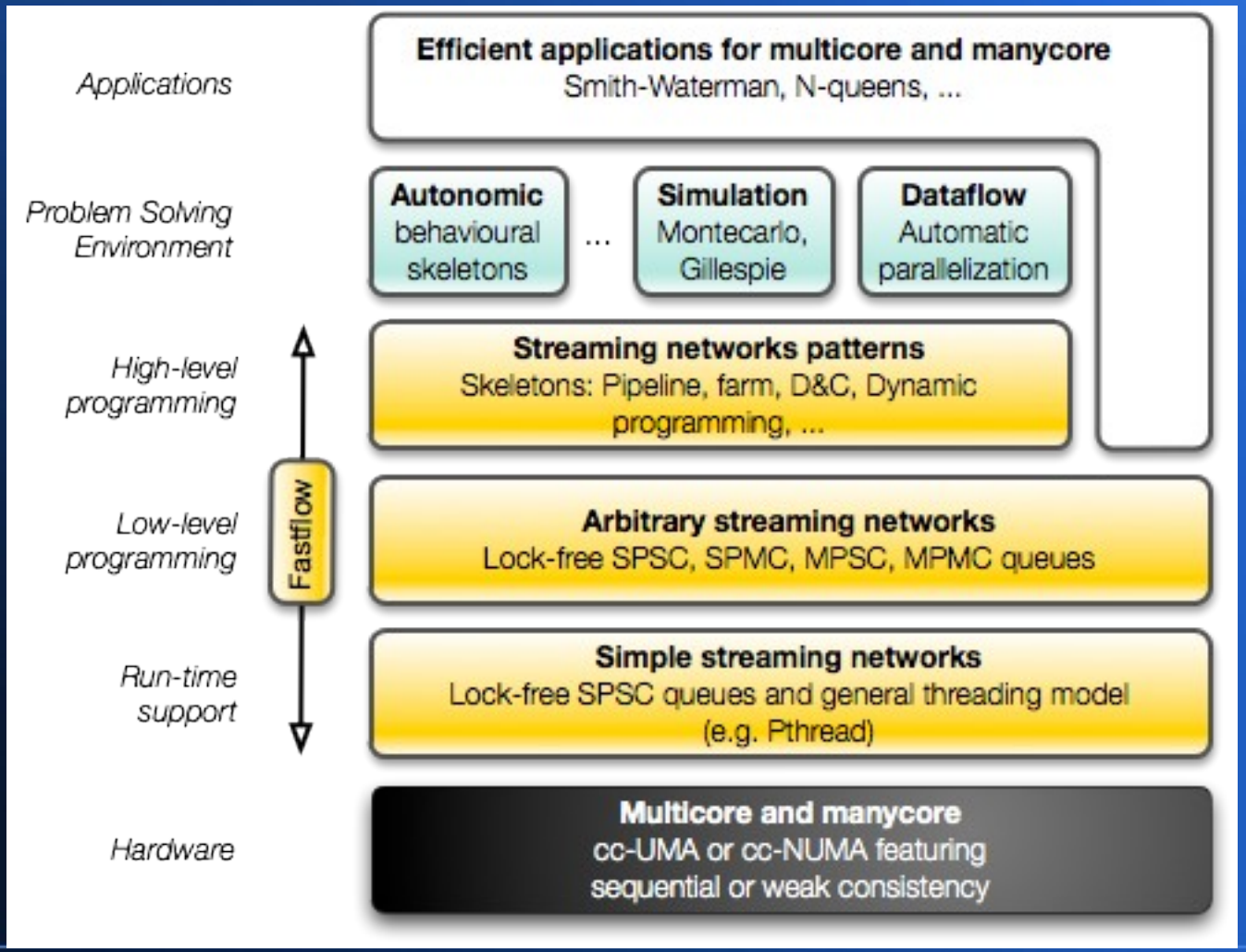
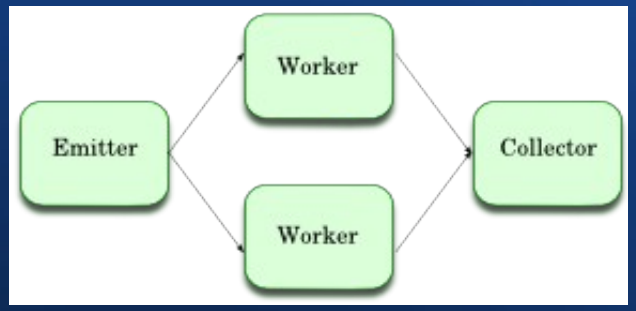
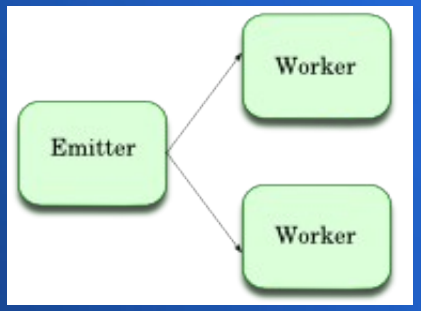
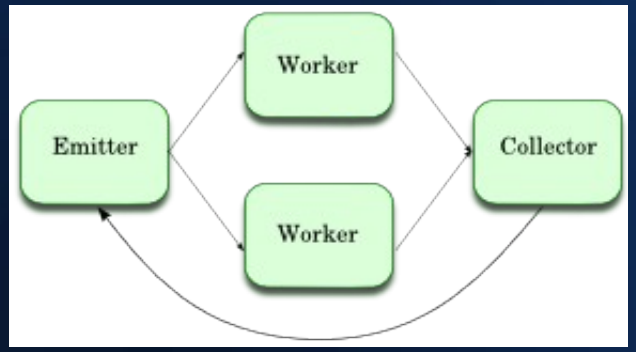


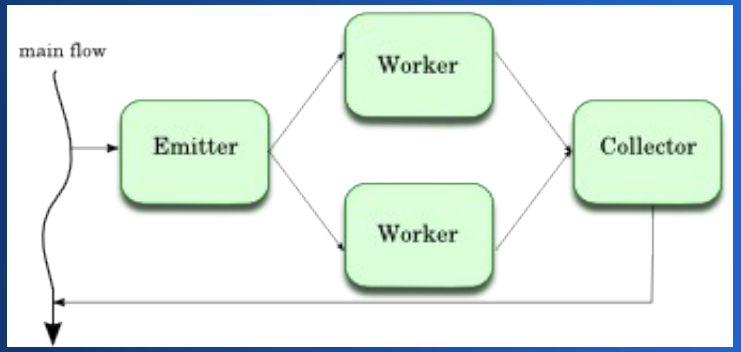
Figure is downloaded from the home-page of FastFlow: http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

Farm schemes



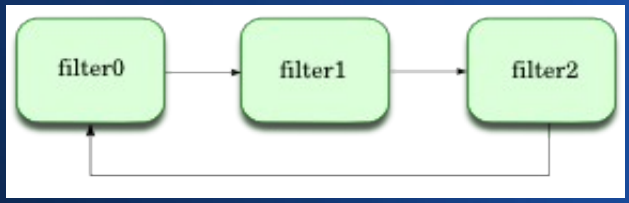


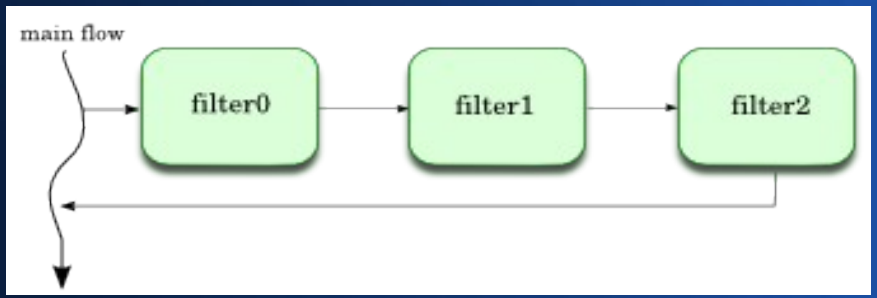




Figures are downloaded from the home-page of FastFlow: http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

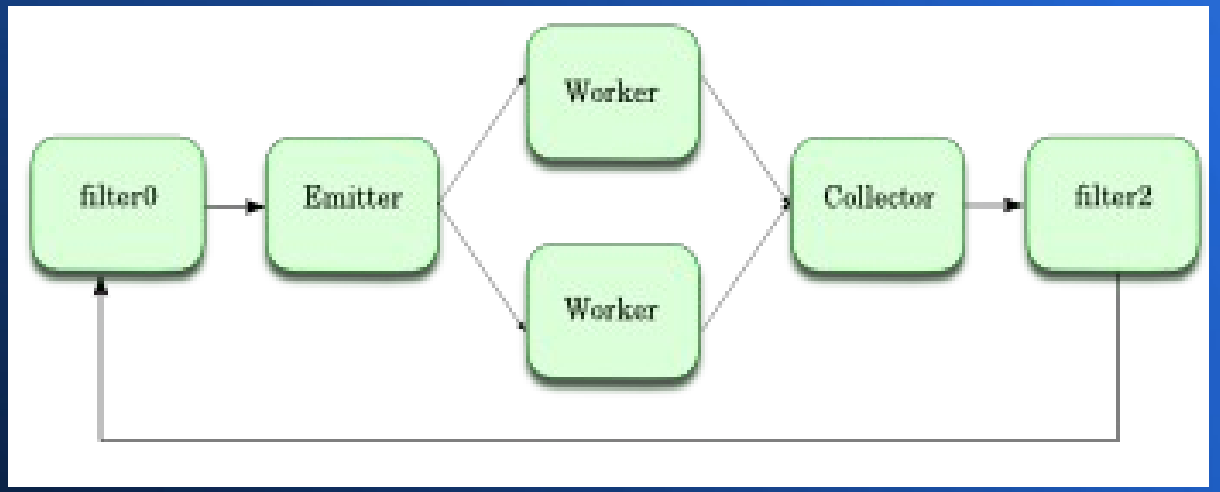
Pipeline schemes

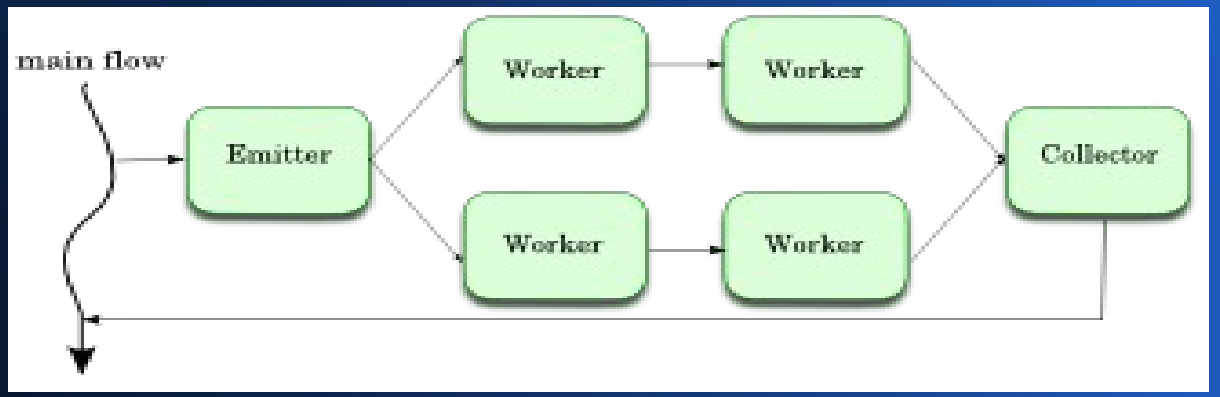




Figures are downloaded from the home-page of FastFlow: http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

Composition of schemes



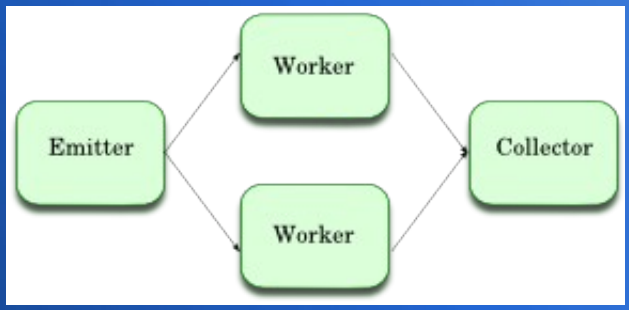


Figures are downloaded from the home-page of FastFlow: http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about

Example

```
struct Emitter:  ff::ff_node
{
    Emitter( int max_task ):ntask( max_task ) {}

    void * svc( void * )
    {
        int * task = new int(ntask);
        --ntask;
        if (ntask<0) return NULL;
        return task;
    }
private:
    int ntask;
};
```




```
struct Collector: ff::ff_node
{
     void * svc( void * task )
     {
          int * t = (int *)task;
          if (*t == -1) return NULL;
          return task;
     }
};
```

```
struct Worker: ff::ff_node
{
    void * svc( void * task )
    {
        int * t = (int *)task;
        std::cout << "Worker " << ff_node::get_my_id()
                  << " received task " << *t << "\n";
        return task;
    }

};
```

Creation of the farm

```
int main()
{
    ff_farm<> farm;

    Emitter E(streamlen);
    farm.add_emitter(&E);

    std::vector<ff_node *> w;
    for(int i=0; i<nworkers; ++i)
        w.push_back(new Worker);
    farm.add_workers(w);

    Collector C;
    farm.add_collector(&C);

    farm.run_and_wait_end();
}
```

Structure of FastFlow

- The base class is ff_node
- In ff_node:

```
virtual void* svc(void* task) = 0;
```

- svc is overridden
- while task is void* → svc is able to capture arbitrary argument in any type

Problem

- void* is needed to cast to a proper type by the programmer → safety deficiency

```
struct Emitter : ff_node
{
   void* svc(void*)
   {
      int* t = new int(ntask);
      /* ... */
      return t;
   }
}
```

```
struct Worker : ff_node
{
   void* svc(void* task)
   {
      double* d = (double*)task;
      /* ... */
      std::cout << *d; //!!!
      return d;
   }
}
```

Why is it void*

- FastFlow is designed by OO Paradigm
- Based on dynamic polymorphism
- svc is virtual function
- svc needs to handle any type of input data
- Function template cannot virtual

Solution proposal

- Apply static polymorphism instead of dynamic polymorphism.
  - proper function is chosen in compile-time
    - improve efficiency
  - no virtual method table
  - no virtual function
  - function can be template
    - argument type deduction is done by the compiler
    - no explicit cast is needed

Curiously recurring template pattern

```
template <typename derived>
struct base
{
    void f()
    {
        static_cast<derived*>(this)->f();
    }
};

struct derived : base<derived>
{
    void f() { /* ... */ }
};
```

Restrictions

- Static polymorphism: proper function call is selected in compile time
- In our solution the farm or pipeline must be defined statically

CRTP in FastFlow

```
template <typename derived>
struct ff_node
{
    template<typename T>
    T* svc(T* task)
    {
        static_cast<derived*>(this)->f(task);
    }
};
```

```
struct Worker : ff_node<Worker>
{
    template<typename T>
    T* svc(T* task)
    {
        std::cout << *d;
        return d;
    }
};
```

Problems with this solution

- There is no common base class:
  - ff_node<Emitter>
  - ff_node<Worker>
  - ff_node<Collector>

  are different types
- The original FastFlow implementation is based on a common base class (refers the parts via ff_node*)
- Buffers stores data via void*

Buffers

- Buffers act as a channel between worker threads
- Each ff_node has own buffer
- Buffers can be either circular or non-circular
- Preprocessor macro defines which kind of buffer to be to use
- Buffers store data via void*

Reimplementing the buffers

- Data type must be a template argument of ff_node instead of member function svc
  - > buffers can be templates
- Replacing preprocessor macros with template metaprograms:

```
#if defined(FF_BOUNDED_BUFFER)
#define FFBUFFER SWSR_Ptr_Buffer
#else  // non-circular buffer
#define FFBUFFER uSWSR_Ptr_Buffer
#endif

FFBUFFER buffer;
```

```
if_<is_circular,
    SWSR_Ptr_Buffer<T>,
    uSWSR_Ptr_Buffer<T> >::type buffer;
```

meta if_

```
template< bool condition, class then_, class else_ >
struct if_
{
    typedef then_ type;
};

template< class then_, class else_ >
struct if_<false, then_, else_>
{
    typedef else_ type;
}
```

Farming with template ff_nodes

- ff_farm is a class represents a farm skeleton
- it has pointers to different ff_nodes
  - Emitter, Worker, Collector
  - While these have no common base, ff_farm needs their type as template argument
- ff_farm is derived from ff_node

The reimplemented ff_farm

```
template<typename E,  // Emitter
         typename W,  // Worker
         typename C,  // Collector
         typename T,  // Data type
         bool is_circular = false,
         typename lb=ff_loadbalancer<E, W, C, T>,
         typename gt=ff_gatherer<C, W, T> >
class ff_farm:
    public ff_node<ff_farm<E, W, C, T, is_circular, lb, gt>, T>;
```

Dummy type for farm schemes

- There are farm schemes which do not requires eg. Collector
- Instantiate ff_farm with special Void type

```
template<class T>
struct Void : ff::ff_node<Void, T>
{
    T* svc(T* task)
    {
        return 0;
    }
};
```

Example

```
// the generic worker
struct Worker: ff::ff_node<Worker, int>
{
    int * svc( int * task )
    {
        std::cout << "Worker "
                  << ff_node<Worker, int>::get_my_id()
                  << " received task " << *task << "\n";
        return task;
    }
};
```

```
// the load-balancer filter
struct Emitter: ff::ff_node<Emitter, int>
{
    Emitter( int max_task ):ntask( max_task ) {}

    int * svc( int * )
    {
        int * task = new int( ntask );
        --ntask;
        if ( ntask<0 ) return NULL;
        return task;
    }
private:
    int ntask;
};
```

```
// the gatherer filter
struct Collector: ff::ff_node<Collector, int>
{
    int * svc( int * task )
    {
        if ( *task == -1 ) return NULL;
        return task;
    }
};
```

Example

```
int main()
{
    ff_farm<Emitter, Worker, Collector, int> farm;
    Emitter e( TASK_NUM );
    Collector c;
    std::vector<ff::ff_node<Worker, int> *> workers;
    for( int i = 0; i < WORKER_NUM; ++i )
        workers.push_back( new Worker() );
    farm.add_emitter( &e );
    farm.add_workers( workers );
    farm.add_collector( &c );

    farm.run_and_wait_end();
}
```

Pipeline

- Stages are derived from ff_node
- Stored in a list of ff_node*
- Handling stages:

```
for(int i=0;i<nstages;++i)
{
    nodes_list[i]->set_id(i);
    nodes_list[i]->run()
}
```

- Problem: type ff_node<Stage1> and
  type ff_node<Stage2> are different

Solution

```
struct Stage1 : ff_node<Stage1, my_data_type>;
struct Stage2 : ff_node<Stage2, my_data_type>;
//...
typedef boost::mpl::vector<Stage1*, Stage2*>::type stage_types;
ff_pipeline<stage_types, my_data_type>* pipe =
     new ff_pipeline<stage_types, my_data_type>();
pipe->add_stage(new Stage1());
pipe->add_stage(new Stage2());
```

Implementation pattern of ff_pipeline

```
template<class typelist, class T>
struct ff_pipeline
{
    std::vector<void*> stages;

    void do()
    {
        do_aux<
            0,
            boost::mpl::size<typelist>::value
        >::do(stages);
    }

    //...
```

do_aux

```
template<int I, int N>
struct do_aux
{
    static void do(std::vector<void*>& stages)
    {
        static_cast<
            typename boost::mpl::at<
                typelist,
                boost::mpl::int_<I>
            >::type
        >(stages[I])->do();

        do_stages_aux<
            I+1,
            boost::mpl::size<typelist>::value
        >::do(stages);
    }
};
```

```
template<int N>
struct do_aux<N, N> /* specialization */
{
    static void do(std::vector<void*>&) { }
};
```

Conclusion

- FastFlow the parallel programming framework for multi-core platforms
- Safety deficiency: void*
- Reimplemented
  - template arguments
  - CRTP instead of virtual functions
  - metaprogram instead of preprocessor macros
- Safer and bit more efficient version of FastFlow

Thank You for Your Attention!

Question?

Zalán Szűgyi, Márk Török, Norbert Pataki
{lupin, tmark}@inf.elte.hu, patakino@elte.hu