Skeleton programming environments
Muesli (1)

Patrizio Dazzi
ISTI - CNR
Pisa Research Campus
mail: patrizio.dazzi@isti.cnr.it

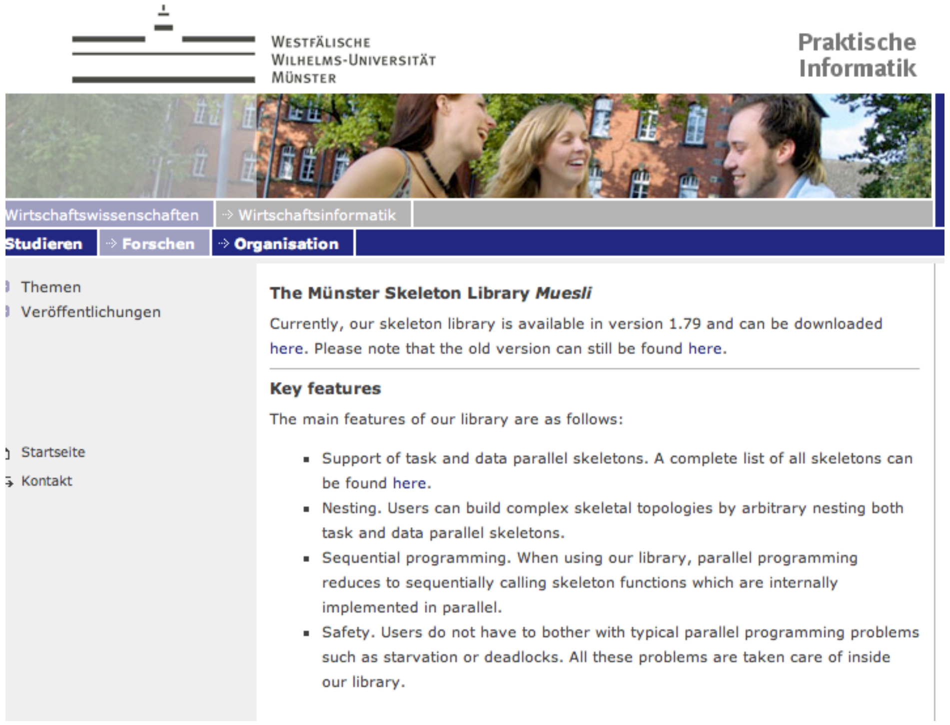
Master Degree (Laurea Magistrale) in
Computer Science and Networking
Academic Year 2009-2010

Outline

- **Muesli Skeleton Library**
- **Muesli Skeletons**
  - *Both for Task and Data Parallel Applications*
- **Using Muesli Skeletons**

Meusli

- **Framework maintained by Herbert Kuchen**
  - *Muenster university, Germany*
- **Originated from previous work on Skil**
  - *with Botorog 1996*
- **C++ based framework**
  - *targeting MPI virtual machine*
  - *intensive usage of templates and peculiar C++ features to implement a user friendly skeleton environment*



The Münster Skeleton Library *Muesli*

Currently, our skeleton library is available in version 1.79 and can be downloaded here. Please note that the old version can still be found here.

Key features

The main features of our library are as follows:

- Support of task and data parallel skeletons. A complete list of all skeletons can be found here.
- Nesting. Users can build complex skeletal topologies by arbitrary nesting both task and data parallel skeletons.
- Sequential programming. When using our library, parallel programming reduces to sequentially calling skeleton functions which are internally implemented in parallel.
- Safety. Users do not have to bother with typical parallel programming problems such as starvation or deadlocks. All these problems are taken care of inside our library.

- **Home page at:** **http://www.wi.uni-muenster.de/pi/forschung/Skeletons/index.html**

Muesli Basic Ideas (1)

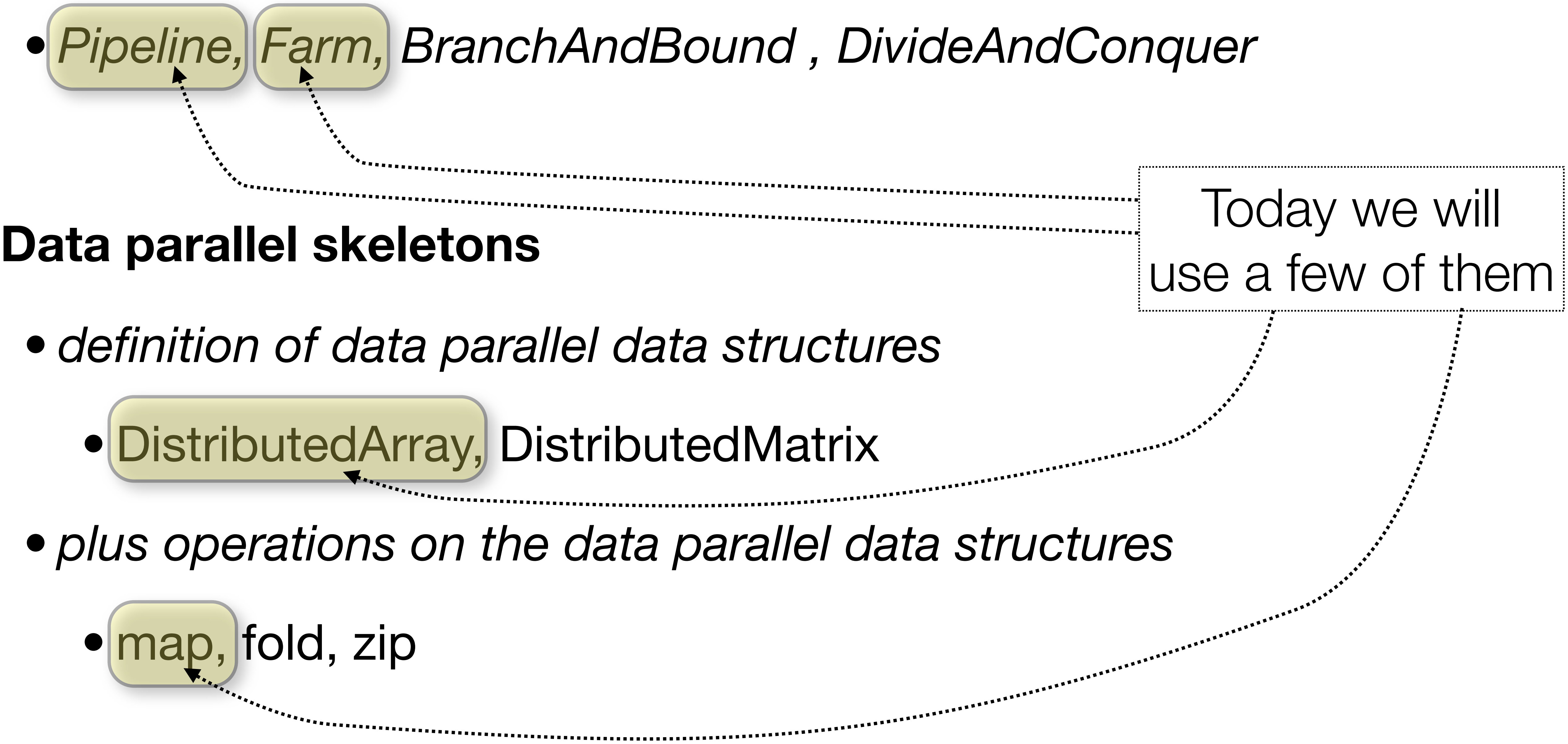
- **Two tier model:**
  - *a parallel computation consists of a sequence of independent task parallel computations.*
  - *an atomic task parallel computation can contain data parallelism*

- **Task parallel computations proceed in two steps:**
  - *a process topology is generated by nesting proper constructors*
  - *the outermost skeleton is started*

Muesli Basic Ideas (2)

- **Data parallel computations use one or more distributed data structures applying data parallel skeletons to them.**
- **sequential computations within a atomic task parallel computation:**
  - *are replicated by all processors participating in the computation.*
  - *an atomic task parallel computation can be seen as a sequential computation*
  - *the operations on distributed data structures happen to have parallel implementations.*

Skeletons in Muesli



- **Stream parallel skeletons**
  - *Pipeline, Farm, BranchAndBound , DivideAndConquer*
- **Data parallel skeletons**
  - *definition of data parallel data structures*
    - DistributedArray, DistributedMatrix
  - *plus operations on the data parallel data structures*
    - map, fold, zip

Today we will use a few of them

“Special” Sequential Skeletons (1)

- **`Initial<T>`**
  - *defined for providing the application input data with type T**
  - *does not take input data*
- **`Atomic<T1, T2>`**
  - *defined to be a classic Pipeline stage or Farm worker*
  - *takes input data with type T1*, provides output data with type T2**
- **`Final<T>`**
  - *defined for managing the application output data with type T**
  - *does not provide output data*

“Special” Sequential Skeletons (2)

- **How to implement the functional code of:**
  - *An Initial skeleton*

```
int current = 10;

int* init(Empty){
  int* i = (int*) malloc(1*sizeof(int));
  *i = current;
  current--;
  if(current < 0) return NULL;
  return i;
}
...
int main(int argc, char** argv){
  ...
   Initial<int> in(init);
   ...
}
```

“Special” Sequential Skeletons (3)

- **How to implement the functional code of:**
  - *An Atomic skeleton*

```
int* compute (int* input){
  cout << "Compute received: " << *input;
  int total = 1;
  for(int i=0; i<*input;i++){
        total *= 2;
  }
  *input = total;
  cout << " - Compute is sending: " << *input << endl;
  return input;
}
...
int main(int argc, char** argv){
  ...
  Atomic<int,int> atomic(compute, 1);
  ...
}
```

“Special” Sequential Skeletons (4)

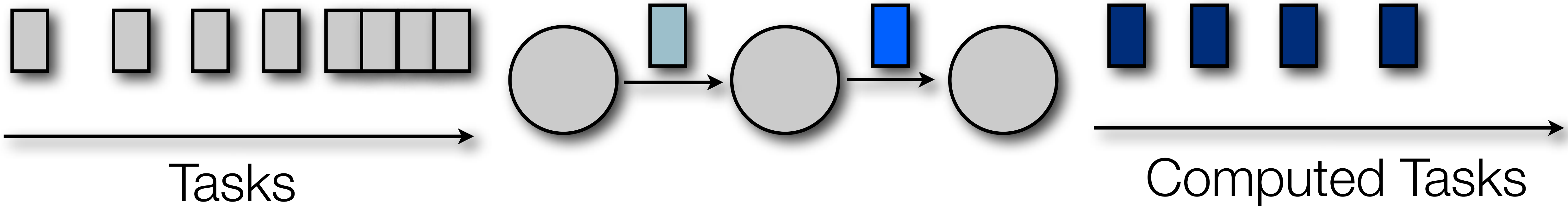
- **How to implement the functional code of:**
  - *An Final skeleton*

```
void fin(int* input){
  cout<< "OUT received: "<< *input << endl;
  free(input);
}

...
int main(int argc, char** argv){
  ...
   Final<int> out(fin);
   ...
}
```

Pipeline in Muesli (1)

- Applications organized in Stages

- Each Stage performs a specific computation



Pipeline in Muesli (2)

- **Class Pipe**

```
class Pipe: public Process
```

- **Two Constructors: 2 and 3 parameters**

```
Pipe(Process& p1, Process& p2): Process();
Pipe(Process& p1, Process& p2, Process& p3);
```

- **Five Methods**

```
void setSuccessors(ProcessorNo* drn, int len);
void setPredecessors(ProcessorNo* src, int len);

void start();

Process* copy();
void show();
```

Pipeline in Muesli (3)

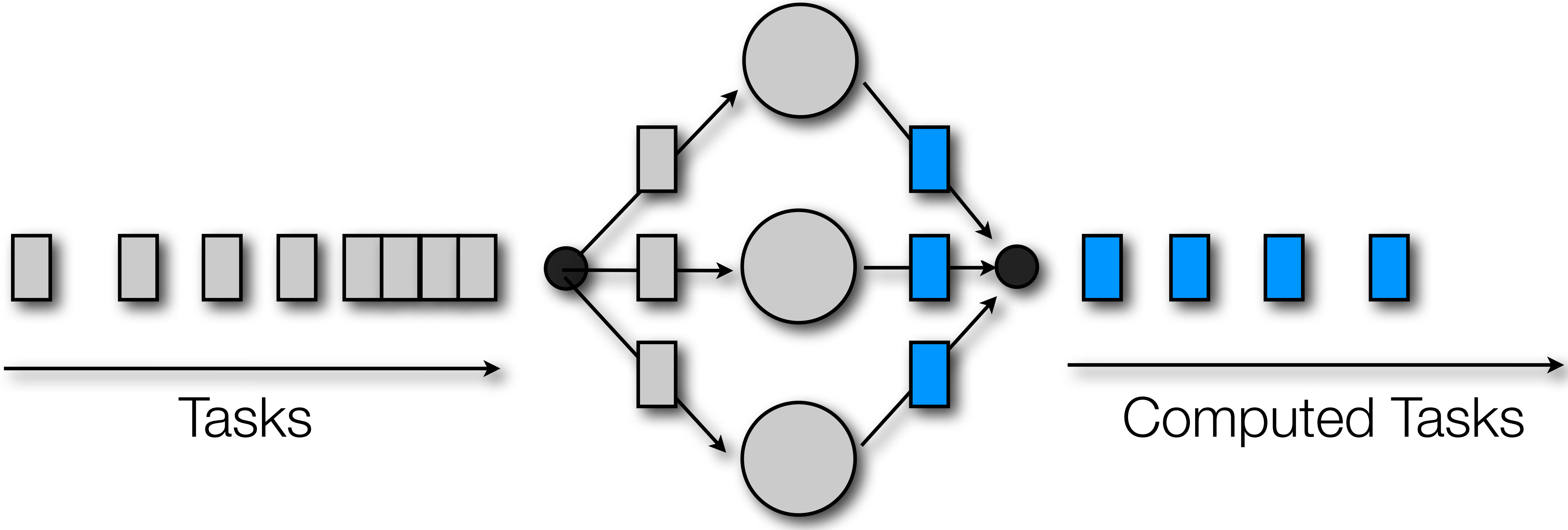
- **Typical instantiation**

```
Initial<int> in(init);
Atomic<int, int> atomic(fun,1);
Final<int> out(fin);

Pipe pipe(in, atomic, out);
```

- **Where:**
  - `Atomic` *is a class creating Stages:*
    - taking input data
    - providing output data
  - `fun` *is a function elaborating data*
  - `init` *is a function generating data*
  - `fin` *is a function printing data*

Farm in Muesli (1)

- Elaborations performed by multiple “workers”
- Each worker computes the same application code



Farm in Muesli (2)

- **class Farm**

```
class Farm: public Process
```

- **Constructors**

```
Farm(Process& worker, int l)
```

- **Five Methods**

```
void setSuccessors(ProcessorNo* drn, int len);
void setPredecessors(ProcessorNo* src, int len);

void start();

Process* copy();
void show();
```

Farm in Muesli (3)

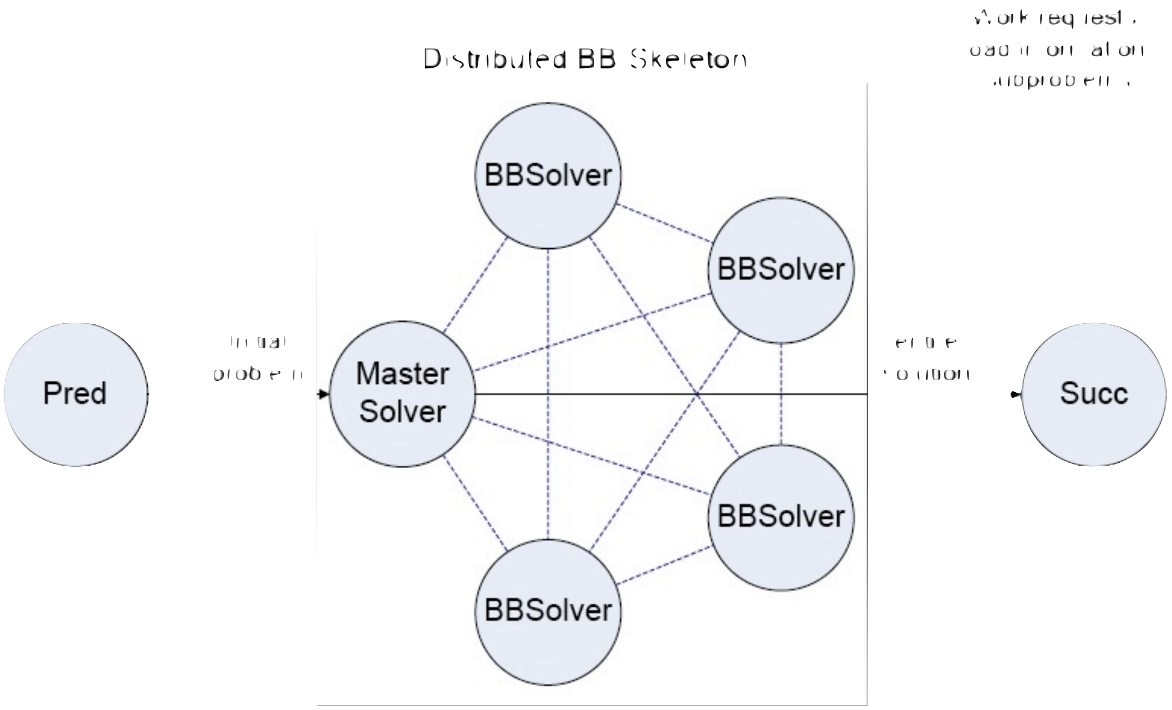
- **Typical instantiation**

```
Atomic<int, int> worker(fun, 1);
Farm<int, int> farm(worker, 3);
```

- **Where:**
  - *`Atomic` is used for creating workers*
  - *`fun` is the function elaborating the input data and providing the output data*

Branch and bound

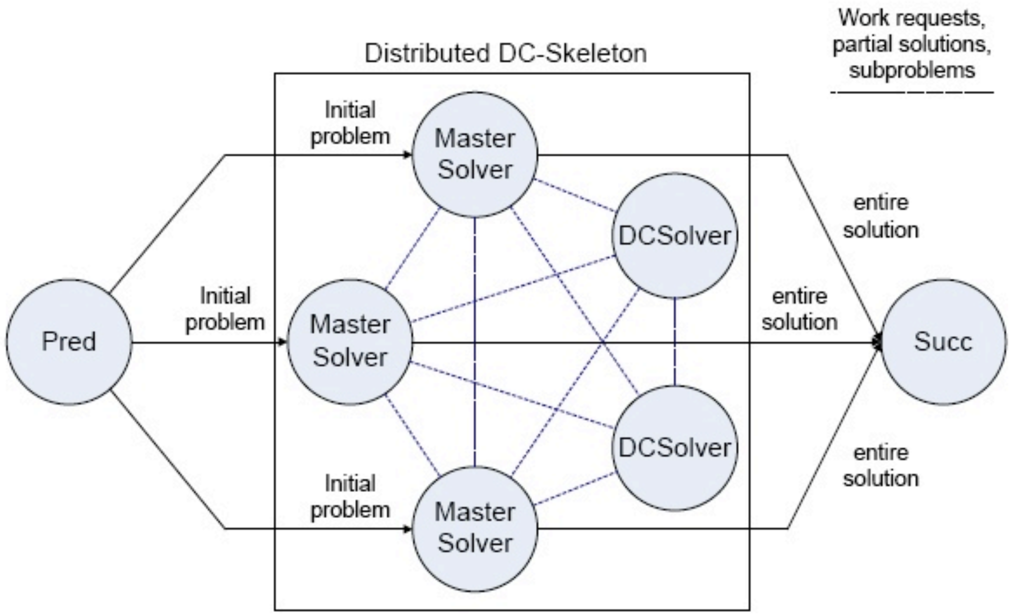
- **Searches the complete solution space of a given problem for the best solution.**
- **Assuming:**
  - *explicit enumeration often impossible in practice.*
  - *the knowledge about the currently best solution.*
  - *the use of bounds to allow to the algorithm to search parts of the solution space only implicitly.*



- **Initially there is only one subset, namely the complete solution space**
- **During the solution process**
  - *a pool of yet unexplored subsets of the solution space, called the work pool, describes the current status of the search.*

Divide and conquer

- **the solution to a problem is**
  - *obtained by dividing the original problem into smaller subproblems and solving the subproblems recursively.*
- **solutions for the subproblems must be combined to form the final solution of the entire problem.**
- **Examples include various sorting methods such as:**
  - *mergesort and quicksort*



Distributed array

- **The class DistributedArray<E> can be used to distribute an array of length size among processes.**
  - *The number of used processes must divide the size of the distributed array without remainder, i.e. size mod n = 0.*
  - *The number n of used processes must be a power of 2. Otherwise, it is not possible to use several data parallel operators*

Distributed matrix

- **Analogous to distributed arrays, but two dimensional.**
  - *consists of a matrix of equally sized partitions.*
- **Each processor collaborating in a data parallel computation**
  - *gets exactly one partition*
  - *is responsible for all computations corresponding to the elements of these partitions.*
- **When constructing a distributed matrix, the numbers of partitions in a row and in a column are fixed.**
- **Partitions assigned to the collaborating processors row by row**

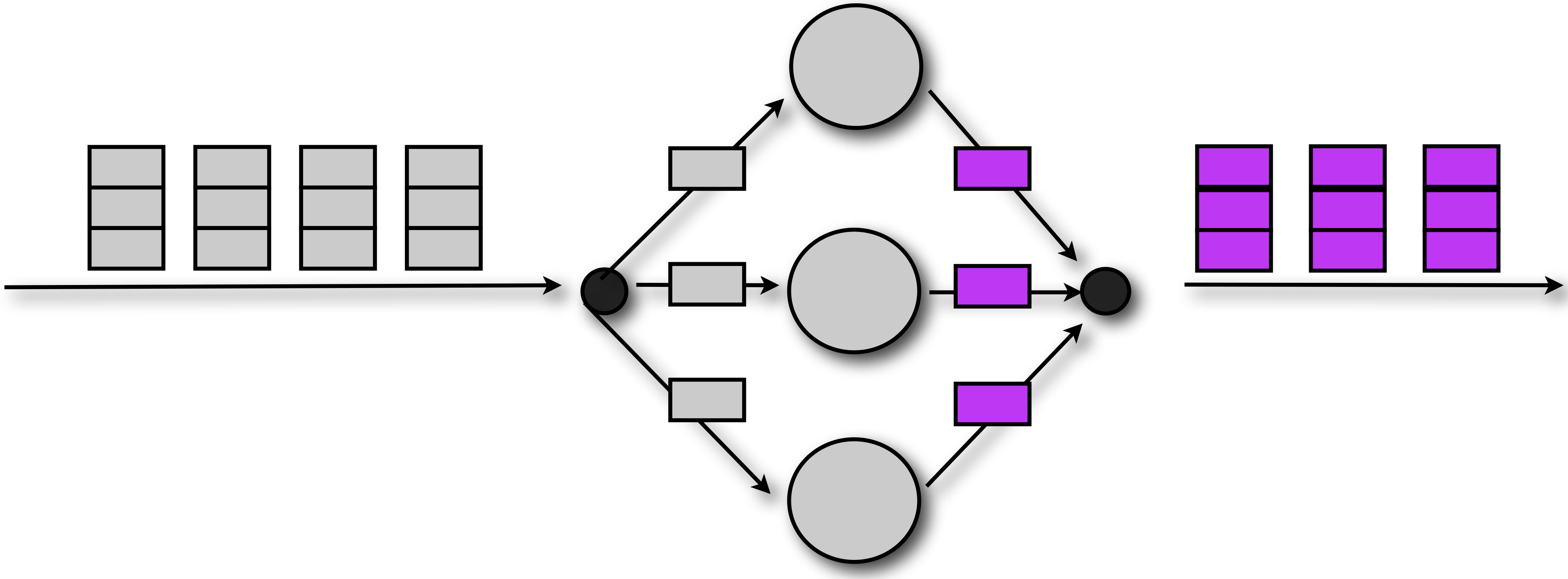
Fold, Zip

- **FOLD: folds the elements of a DistributedArray into a single value by repeatedly applying an associative and commutative binary function to them.**
  - *E.g. sum all the array elements*

- **ZIP: combines corresponding elements (with respect to their index) of two (equally sized) Distributed Arrays using a binary function.**

Map (1)

- **map replaces each element of a distributed data structure by the result of applying a function to it**
  - *variants: mapIndex, mapInPlace, mapIndexInPlace, mapPartitionInPlace*



Map (2)

- **Programming a map in Muesli means:**
  - *defining a Distributed Structure*
  - *computing one of the map function variants on it*

```
DistributedArray<Point> A(1000, &random);
A.mapIndexInPlace(&f);
```

  - *where:*
    - random is the initializer function
    - f is function computed by the map processes

Map (3)

- *Example initializer function:*

```
Point random(int i) {
  int r = rand();
  srand(r + MSL_myId);
  return Point( double(rand()%10), double(rand()%10) );
}
```

- *Example function computed by the map processes:*

```
Point f( int index, Point p ) {
  cout<<"Point at "<< index <<" was " << p << endl;
  p.Add(v);
  cout<<"Point at " << index << " is " << p  << endl;
  return p;
}
```

A few code considerations (1)

- **despite the template argument, the data taken and returned by skeleton functions is a pointer**
  - *E.g. In* `Initial<int> init(func);`

    `func` *needs to be declared as:*

    ```
    int* funct(){
            ...
         return data;
    }
    ```

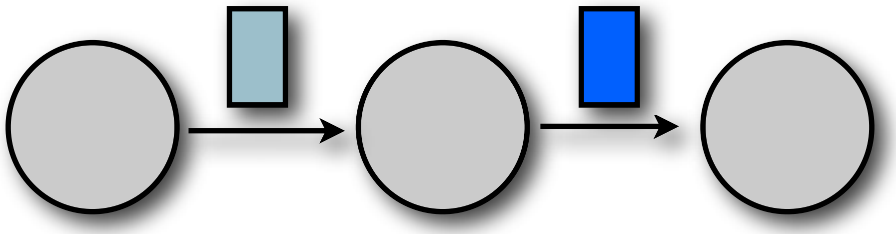
    *where the type of data needs to be int**

A few code considerations (2)

- **Initial Stages to indicate that the input stream is finished has to return NULL**
  - *E.g. In* `Initial<int> init(func);`

```
int* funct(){
        ...
    if(NoMoreInputData) return NULL;
    else return data;
}
```

Sample Pipeline Usage (1)



```
#include "Muesli.h"
#include <iostream>

using namespace std;
int current = 10;
```

Function `init` emits data

```
int* init(Empty){
   int* i = (int*) malloc(1*sizeof(int));
   *i = current;
   current--;
   if(current < 0) return NULL;
   cout << "IN is sending: " << *i << endl;
   return i;
}
```

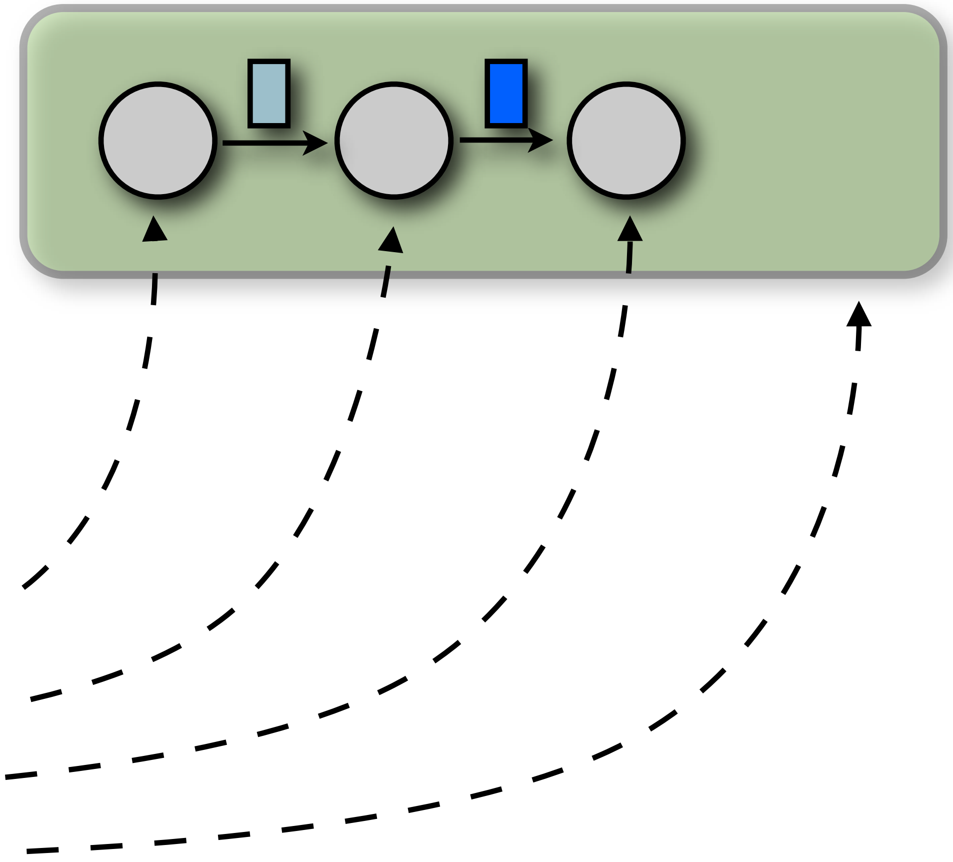
Function `compute` elaborates data

```
int* compute (int* input){
   cout << "Compute received: " << *input;
   int total = 1;
   for(int i=0; i<*input;i++){
           total *= 2;
   }
   *input = total;
   cout << " - Compute is sending: " << *input << endl;
   return input;
}
```

Function `fin` prints out data

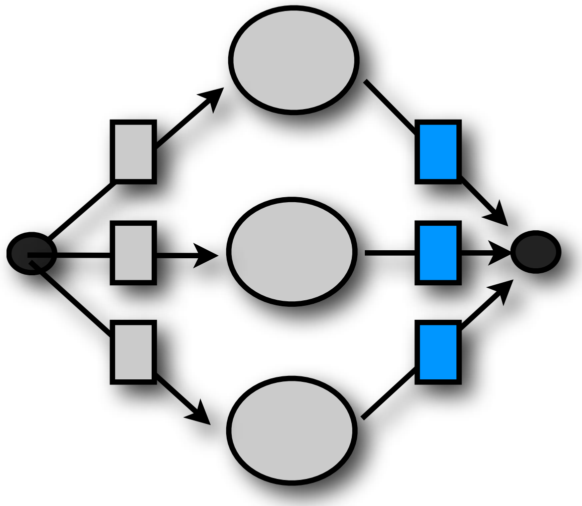
```
void fin(int* input){
   cout<< "OUT received: "<< *input << endl;
   free(input);
}
```

Sample Pipeline Usage (2)



```
int main(int argc, char** argv)
{
   InitSkeletons(argc,argv);

   Initial<int> in(init);
   Atomic<int, int> atomic(compute,1);
   Final<int> out(fin);
   Pipe pipe(in, atomic, out);

   pipe.start();

   TerminateSkeletons();
   return 0;
}
```

Using a Farm as second stage (1)



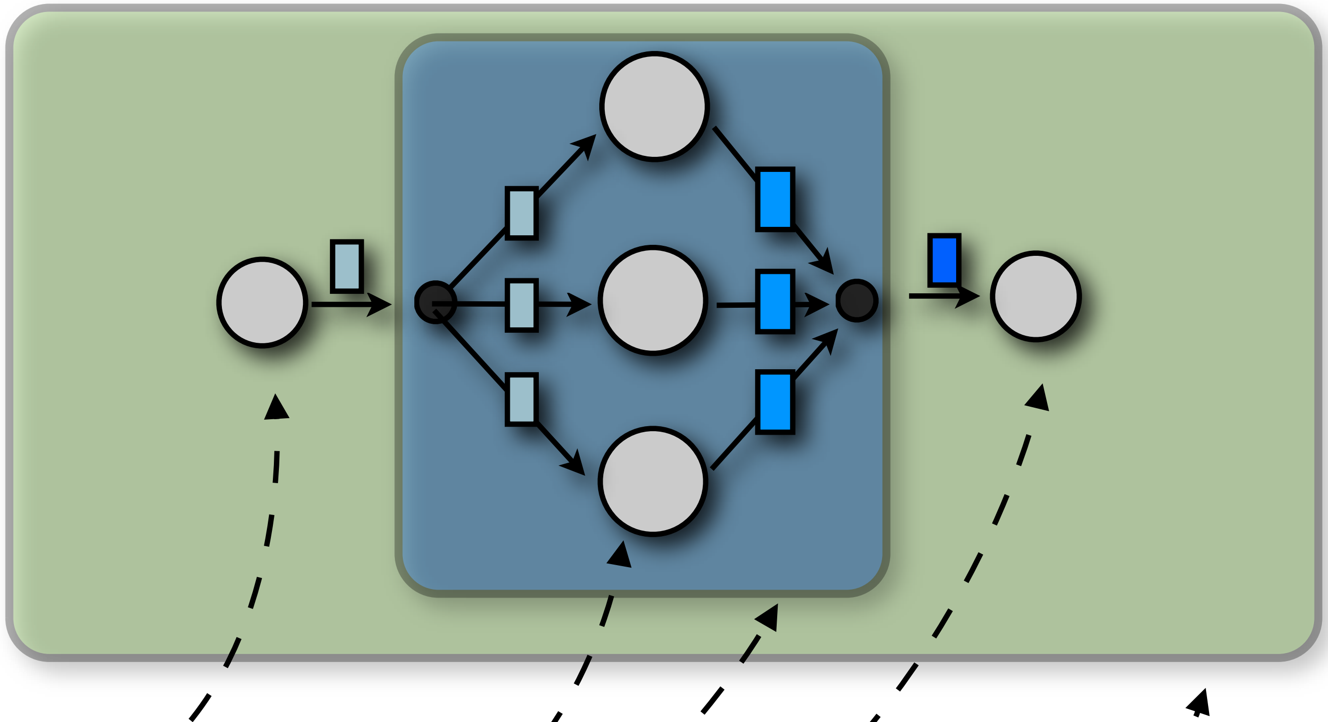
Function `init` emits data

Function `compute` elaborates data

Function `fin` prints out data

```
#include "Muesli.h"
#include <iostream>

using namespace std;
int current = 10;

int* init(Empty){
   int* i = (int*) malloc(1*sizeof(int));
   *i = current;
   current--;
   if(current < 0) return NULL;
   cout << "IN is sending: " << *i << endl;
   return i;
}

int* compute (int* input){
   cout << "Compute received: " << *input;
   int total = 1;
   for(int i=0; i<*input;i++){
           total *= 2;
   }
   *input = total;
   cout << " - Compute is sending: " << *input << endl;
   return input;
}

void fin(int* input){
   cout<< "OUT received: "<< *input << endl;
   free(input);
}
```

Using a Farm as second stage (2)



```
int main(int argc, char** argv)
{
   InitSkeletons(argc,argv);

   Initial<int> in(init);
   Atomic<int, int> atomic(compute,1);
   Farm<int, int> farm(atomic,3);
   Final<int> out(fin);
   Pipe pipe(in, farm, out);

   pipe.start();

   TerminateSkeletons();
   return 0;
}
```

Using Complex Input Data

Extends MSL_Serializable

```
class Point : public MSL_Serializable
{
  double _x;
  double _y;
public:
  Point( double x = 0.0, double y = 0.0 ){ _x = x; _y = y; }
  double& X(){ return _x; }
  double& Y(){ return _y; }
  void Add(const Point& o) { _x += o._x; _y += o._y; }

  inline int getSize() { return sizeof(double) + sizeof(double); }

  void reduce(void* pBuffer, int bufferSize) {
      double* adr1 = (double*) memcpy(pBuffer,&_x,sizeof(double));
      adr1++;
      double* adr2 = (double*) memcpy(adr1,&_y,sizeof(double));
  }
  void expand(void* pBuffer, int bufferSize) {
      double *dd = (double*)pBuffer;
      _x = dd[0];
      _y = dd[1];
  }
};

  std::ostream& operator<<( std::ostream& o, const Point& p ) {
    o<<"(" << p.X() << "," << p.Y() << ")";
    return o;
  }
```

Enabling to print out that complex data

Data Parallel with MapIndexInPlace on a DistributedArray

Map function

Initializer function

```
Point f( int index, Point p ) {
  cout<<"Point at "<< index <<" was " << p << endl;
  p.Add(v);
  cout<<"Point at " << index << " is " << p  << endl;
  return p;
}

Point random(int i) {
  int r = rand();
  srand(r + MSL_myId);
  return Point( double(rand()%10), double(rand()%10) );
}

int main(int argc, char** argv)
{
  InitSkeletons(argc,argv);

  DistributedArray<Point> A(1000, &random);
  A.mapIndexInPlace(&f);

  TerminateSkeletons();
  return 0;
}
```

Questions ?