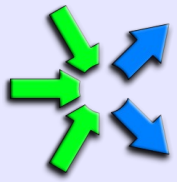
FastFlow introduction (3)



Massimo Torquati

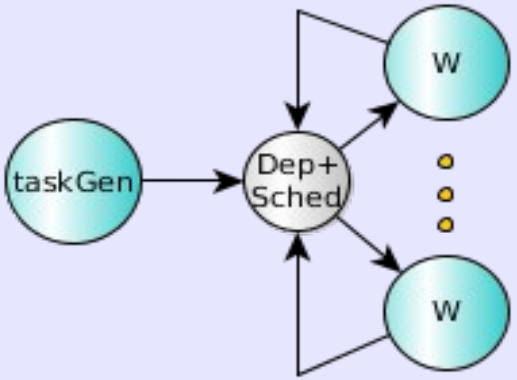
Computer Science Department, University of Pisa – Italy

torquati@di.unipi.it

SPM, 25 November 2014

Data-Flow parallelism

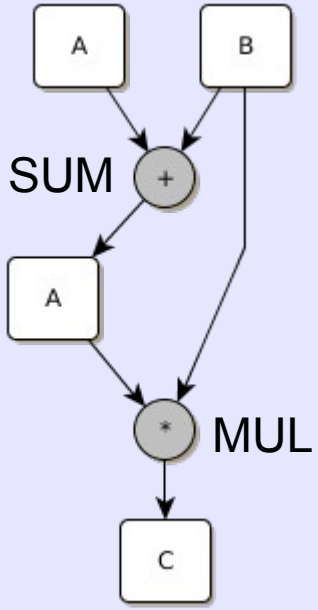
- The **data-flow** programming model is a general approach to parallelism based upon data dependencies among program's operations.
- Computation expressed by the data-flow graph (DAG) whose nodes are instructions and arcs are pure data dependencies (read-after-write dep.).
- **Macro** Data-Flow: is the same concepts but instructions are macro (fat) instructions (entire function or a block of code).
- In FastFlow we have the **ff_mdf** pattern. Currently it is implemented as a 2-stage pipeline whose second stage is a task-farm without collector and with feedback channel.



FastFlow ff_mdf pattern

- The ff_mdf pattern targets macro-data-flow computations
- The class interface is defined in the file *mdf.hpp*
- The user has to specify INPUT and OUTPUT data-dependencies for each task by providing pointers to input and output data
- A task is generated with the method ***AddTask*** *sequentially* respecting the program order.
- The run-time takes care of dependencies and task scheduling

A = A + B;
C = A * B;



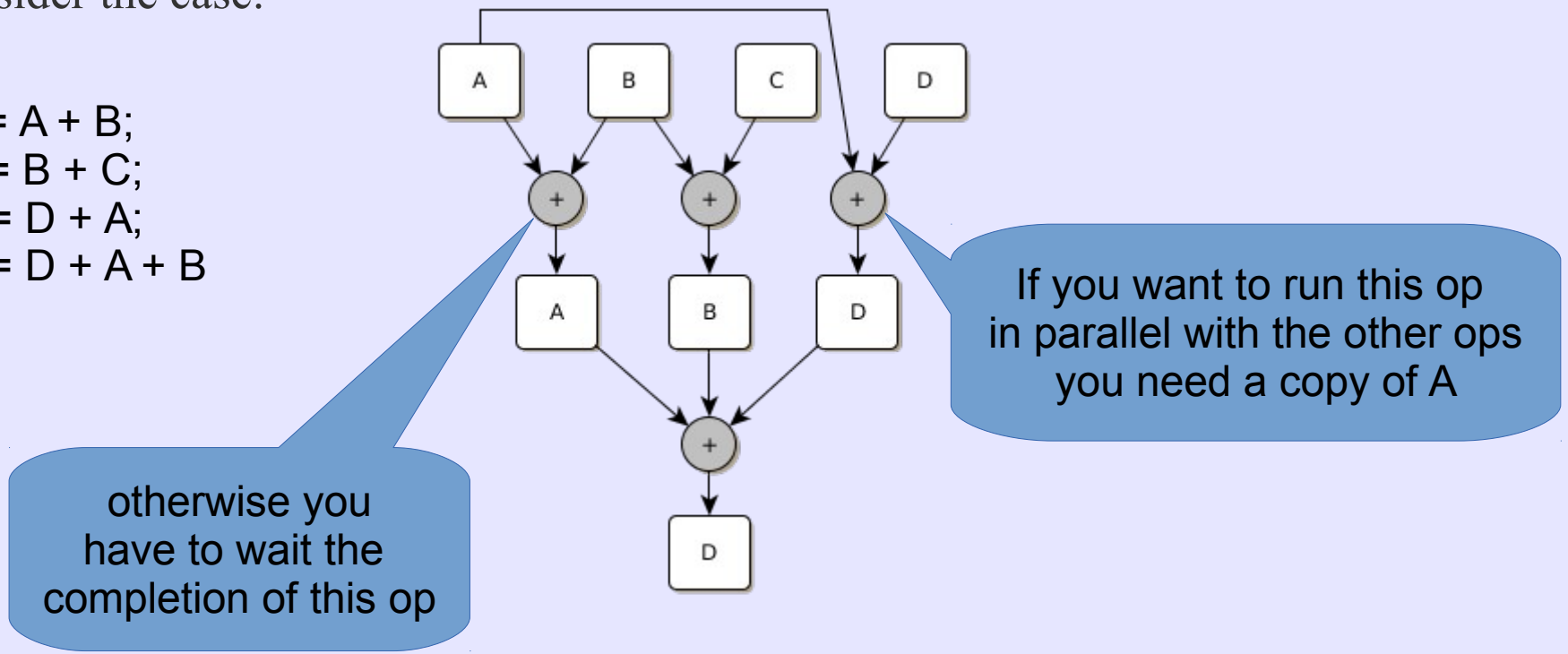

```
// X = X+Y
void SUM(long *X, long *Y, size_t size);
// Z = X*Y
void MUL(long *X, long *Y, long *Z, size_t size);

{ // A = A+B
  const param_info _1={&A, INPUT};
  const param_info _2={&B, INPUT};
  const param_info _3={&A, OUTPUT};
  std::vector<param_info> P={_1,_2,_3};
  mdf->AddTask(P, SUM, A, B, size);
}
{ // C = A*B
  const param_info _1={&A, INPUT};
  const param_info _2={&B, INPUT};
  const param_info _3={&C, OUTPUT};
  std::vector<param_info> P={_1,_2,_3};
  mdf->AddTask(P, MUL, A, B, C, size);
}
```

Data-flow on shared-memory, warning !

- On shared memory, to reduce memory consumption in-place computation is generally used, i.e. A = F(A,...)
- Pay attention because you may have anti-dependencies (write-after-read) that in order to be solved may imply either memory-copy or extra synchronization (less parallelism)
- Consider the case:

```
A = A + B;
B = B + C;
D = D + A;
D = D + A + B
```



ff_mdf example: a simple workflow

- The constructor interface of the ff_mdf pattern is:
  - ff_mdf<T> (taskF, taskF_arg, …<low-level configuration params> ...);
  - taskF is a function getting a pointer to a single argument of type T
  - taskF_arg is of type T

- Let's take a look at the simple program contained in the tutorial named wf.cpp

Task parallelism

- Tasks are units of work that perform a specific job.
- A task can be a function/procedure or a block of code.
- Typically a “big” task can be decomposed into additional, more fine-grained tasks.
- Differently from data-flow parallel executions, task-parallel executions may require explicit synchronisations among tasks (typically global sync. i.e. barriers)
- When using task-parallelism the parallelism is organised around the functions to be executed rather than around the concept of data movement or decomposition.
  - On this respect, pipeline (function decomposition) and task-farm (function replication) can be seen as task-parallel patterns.
- In FastFlow we have the ***ff_taskf*** pattern.
  - Currently it is implemented as a task-farm skeleton without collector and with feedback channel.
  - It schedules functions or lambdas

FastFlow ff_taskf pattern

- The ff_mdf pattern targets scheduling of functions and lambdas.
  - For the moment no support for recursive functions
- The class interface is defined in the file taskf.*hpp*
- A task is generated with the method ***AddTask***
- Simple example:

```
std::vector<long> A(SIZE_A);
std::vector<long> B(SIZE_B);

for(long i=0;i<SIZE_A;++i)
      A[i] = F(A,i);

for(long i=1;i<SIZE_B;++i)
      B[i] = G(B,i);

long x = H(A,B);
```

```
#include<ff/taskf.hpp>
using namespace ff;

std::vector<long> A(SIZE_A);
std::vector<long> B(SIZE_B);

auto Task1 = [&A]() {  for(long i=0;i<SIZE_A;++i)
                           A[i] = F(A,i);};
auto Task2 = [&B]() {  for(long i=1;i<SIZE_B;++i)
                           B[i] = G(B,i); };

ff_taskf tf;
tf.AddTask(Task1);
tf.AddTask(Task2);

tf.run_then_freeze();  // this is a barrier point

long x = H(A,B);
```

ff_taskf example: block-based matmult

- Let's take a look at the example named **blk_matmul_taskf.cpp**