

# Tearing For Parallelization and Control of Sparsity in Process Flowsheeting

Paolo Greppi<sup>a</sup>

<sup>a</sup>*Università degli Studi, Via all'Opera Pia, 15/A, Genova 16145, Italy*

## Abstract

The shift from single- to multi-core hardware in desktop workstations is opening up new ways for parallelizing CAPE tools. In the process flowsheeting domain it gives us a chance to reconsider the topic of tearing; tearing can be used for coarse-grained parallelization and sparsity control of a single-pass modular flowsheet calculation. The idea is demonstrated in the framework of a flowsheeting tool under development based on the object-oriented programming language C++ and the OpenMP multithreading specification. We have provided speed-up estimates based on actual chemical processes as well as prototype test problems.

**Keywords:** schedule, multi-core, task graph.

## 1. Introduction

In the modular approach a process flowsheet can be represented as a DG (Directed Graph) where unit operation models are the nodes while material, energy or information streams are the arcs, with the direction of the arc indicating the direction of flow. The unit operations isolate certain subsets of highly non-linear, implicit equations (such as flashes, reactors, countercurrent separations) in black boxes so that they can be solved separately using special-purpose algorithms.

### 1.1. Tearing

If the DG is a DAG (Directed Acyclic Graph), it can be calculated by substitution in one pass. If there is any recycle, feedback or looping the flowsheet is implicit and it is necessary to identify a set of unknowns capable of transforming the cyclic DG into a DAG (decyclification), initialize them and use a specific algorithm to solve iteratively for the unknowns. The decyclification is accomplished by tearing certain arcs and exposing the corresponding variables as unknowns.

In the “sequential modular” approach the DG is decyclified by tearing a minimal set of material streams; the minimal tearing can be found by solving the minimal FAS (Feedback Arc Set) problem, equivalent to the MAS (Maximum Acyclic Subgraph).

In the “simultaneous modular strategy” approach described by (Kulikov et al. 2005) all material, energy or information streams entering any unit operation become unknowns, while the variables inside the black-box unit-operation models remain invisible to the main solver.

### 1.2. Depth, width and initialization

The decyclified DAG representing the equation set ready for solution can be characterized by its width (i.e. number of unknowns, function of the number of torn arcs) and by its depth (the maximum number of unit operations that have to be chained to compute the residuals).

The width impacts on the unknown initialization and the size of the system of equations; the depth impacts on the sparsity and hence the overhead involved in automatic differentiation. A high width is undesirable at the beginning of the iterative resolution because all unknowns have to be properly guessed (initialized), therefore the technique of minimal tearing is often preferred.

With complete tearing, on the other hand, we obtain a rearranged DAG with low depth (the computation of only one unit operation at most is required to obtain the residual as a function of the unknowns), admitting parallelization but with high width, having the maximum number of unknowns.

### *1.3. Parallel computing*

According to hardware manufacturers the transition from workstations based on a single core CPU (Central Processing Unit) to multi-core will characterize the next decades of desktop computing. The shared-memory, SMP (Symmetric Multi-Processing) model of parallel computing characteristic of these architectures will push the software industry to migrate toward multithreaded applications that exploit it.

Most research in the past has been on developing number crunching algorithms specific for different multiprocessing architectures such as vector and SIMD (Single Instruction Multiple Data), Message Passing Interface (MPI), distributed memory and massively parallel systems - see for example (Camarda et al. 1999) and a few works in the domain of parallel simulation of Very-Large-Scale Integration (VLSI) electronic chips: (Wu 1976), (Mi-Chang Chang et al., 1988). There is a lack of research on parallelizing the algorithms of process flowsheeting on shared-memory architectures with a small number of parallel threads. We will show that by choosing intermediate positions between the extremes of minimal versus complete tearing it is possible to use tearing to control the depth (and hence the sparsity) and/or to rearrange the graph so that the work load can be effectively scheduled on parallel execution cores.

## **2. Enhancement of parallelism and tearing**

This contribution focuses on the single-pass calculation of the flowsheet, without considering the algorithms required for the actual resolution, nor the process-specific computations performed.

### *2.1. Task Graphs*

For the purpose of scheduling, the process flowsheet can be interpreted as a task graph (sometimes called a Task Precedence Graph, TPG); a task graph is a graph representing a partial ordering, where each node is a task to perform, and an arc directed from one node to another is a precedence constraint so that the execution of the former task precedes that of the latter task.

Weights associated with the nodes and edges represent the computation and communication costs, respectively.

Typically, one entry and one exit dummy node (“source” and “sink” respectively) with zero computation cost are used to represent the starting point and the end point of the tasks.

### *2.2. Scheduling*

The objective of scheduling is to minimize the completion time (or make span) of the parallel execution of a task graph by allocating the tasks to the processors (or cores) while respecting the precedence constraints. This problem has been studied extensively

due to its applications to project management, optimizing compilers and information processing and is known to belong to the class of "strong" NP-hard problems.

In the case of flowsheet scheduling on a shared-memory parallel computer the communication costs (i.e. the cost of "executing" the arcs) can be neglected. This is not the case for MPI parallel architectures, (Aronsson et al. 2002) where task duplication is employed to reduce communication costs.

In our case it is possible to pick up a scheduling algorithm from the "Task Precedence Graph with arbitrary structure and computational costs and no communication" branch of the taxonomy presented in the review paper (Kwok et al. 1999).

In particular the heuristic algorithm CP/MISF (Critical Path/Most Immediate Successors First) (Kasahara et al. 1984) has been used because it is both simple and powerful.

### 2.3. Critical path and parallelization

Even with an infinite number of available processors the make span can never be shorter than the time required to execute the tasks lying along the CP (Critical Path), i.e. the longest path from the source to the sink nodes. The CP can be reduced by tearing; (Sandnes et al. 2005) highlight that decyclification of the cyclic DG into a DAG can be performed by tearing (selecting a "start node") in various manners, yielding DAGs with different critical path lengths and different make spans.

In our work, on the other hand, we have always chosen to decyclify using the minimal-tearing obtained by solving the minimal FAS/MAS problem, and successively reduce the cost to traverse the CP by tearing additional arcs.

## 3. Implementation

The algorithms have been implemented within LIBPF (Greppi 2006), a novel simulation tool based on the object-oriented programming language C++ and using the OpenMP multithreading specification.

The OpenMP (Open specifications for Multi Processing) is a specification (OpenMP 2005) that can be used to specify shared memory parallelism in FORTRAN and C/C++ programs. OpenMP has been selected because it is a *de-facto* standard and supported by most compilers today; it is more portable and operates at a higher-level than the operating-system level (POSIX or Win32) primitives, it is very lightweight and non-intrusive. The price to pay is that OpenMP leaves the responsibility for identifying regions of code suitable to parallelization and preventing conflicts, races and deadlocks to the application developer.

### 3.1. Scheduling

The iterative tearing / parallel scheduling algorithm is based on a target make span.

A pseudo-code of the algorithm is:

---

```

estimate the costs (execution times) of all tasks
select a target make span
do {
  schedule the task graph using CP/MISF
  identify one arc a, candidate for tearing
  if not(trivial(a)) tear a
} while not(trivial(a))

```

---

If the objective is to optimize the parallel execution, the target make span is in the range of the ideal make span, the latter being the total cost (sum of all the costs of the nodes)

divided by the number of parallel cores. We have found that a good choice for the target make span is the ideal make span increased by a fraction (i.e. one half) of the average cost of a node.

In case the objective is to control the depth, the target make span is set to the target depth multiplied by the average cost of the nodes.

To select the candidate arc for tearing, the priority list, in descending order of level, which is built during the CP/MISF algorithm, is examined, looking at the starting time of each task according to the current schedule. When a task that has a high level but is currently scheduled to start at a later time than the target make span is encountered, the controlling precedence relationship that prevents this task from starting earlier is our candidate arc for tearing.

A candidate arc is considered trivial if its tearing brings no benefit, i.e. if it is either from source or to sink.

### 3.2. Execution

Once an optimized parallel schedule for a single flowsheet pass is available, it is possible to execute it, but since the real task execution times will differ from the estimated ones, it is necessary to set up a “simple lock” (OpenMP 2005) in each arc. These locks will be initialized and set by the upstream node before the start of the execution. Each node tries to reserve the locks for all incoming arcs when it is requested to execute; if any of these is still owned by another thread, the current thread is blocked until the lock is available. At the end of the execution the node frees the locks on all downstream arcs.

A pseudo-code of the execution procedure, showing the OpenMP compiler directives necessary for parallelization, is:

---

```
#pragma omp parallel
{
  #pragma omp for
  for each thread
    set outgoing locks
  #pragma omp barrier
  #pragma omp for
  for each thread {
    for each task assigned to this thread {
      try to set incoming locks
      execute
      release outgoing locks
    } // for each task
  } // for each thread
} // omp parallel
```

---

## 4. Results

The results are based on scheduling on 4 processors. This figure has been chosen since it is the typically available number of cores on today’s workstation CPUs.

For illustrative purposes the first set of results is based on the toy task graph of Figure 1. The flowsheet is already a DAG so it can be executed on a sequential processor,

requiring 16 time units. It can also be scheduled on four processors resulting in a make span of 7 time units as shown on the first column.

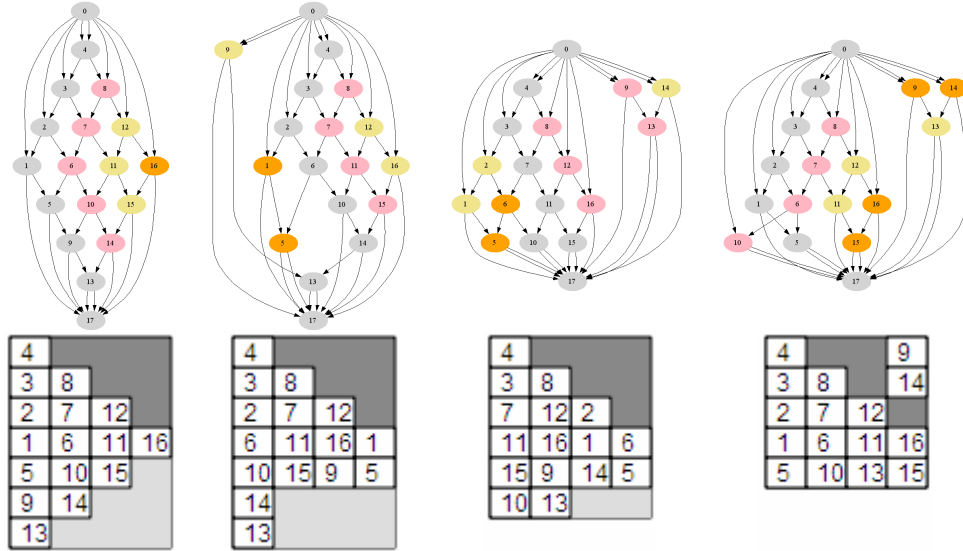


Figure 1 - Scheduling a 4x4 cross-flow arrangement of 16 nodes of equal cost (1 time unit)

The proposed algorithm decides to tear the 5 arcs 5→9, 10→9, 10→14, 15→14 and 11→10 reducing the make span to 5. The ratio of the estimated execution time of the sequential single-processor algorithm to the parallel span time (the estimated execution time of the parallel algorithm) is the speed-up. The tearing of an additional 5 arcs decreases the make span from 5 to 7 and consequently increases the speed-up from 228% to 320%, bringing it closer to the ideal speed-up of 400%.

The algorithm was also tested on more significant task graphs; results are shown on Table 1.

| Test case       | nodes | arcs | arc/<br>node | minimal<br>tearing | speed-up | new torn<br>arcs | new<br>speed-up |
|-----------------|-------|------|--------------|--------------------|----------|------------------|-----------------|
| fpppp.stg       | 334   | 1145 | 3.4          | 0                  | 394 %    | 0                | =               |
| rand0042.stg    | 100   | 301  | 3            | 0                  | 374 %    | 0                | =               |
| rand0066.stg    | 100   | 410  | 4.1          | 0                  | 369 %    | 0                | =               |
| robot.stg       | 88    | 131  | 1.5          | 0                  | 349%     | 5                | 360%            |
| sparse.stg      | 96    | 67   | 0.7          | 0                  | 392%     | 0                | =               |
| cross_flow      | 16    | 40   | 2.5          | 0                  | 228%     | 5                | 320%            |
| mcfc            | 23    | 34   | 1.5          | 4                  | 141%     | 5                | 262%            |
| multistage_evap | 25    | 49   | 2            | 4                  | 111%     | 12               | 296%            |
| gas_compr       | 100   | 116  | 1.2          | 0                  | 241%     | 4                | 393%            |
| chem1           | 148   | 156  | 1.1          | 12                 | 376%     | 0                | =               |
| chem2           | 58    | 102  | 1.8          | 1                  | 131%     | 7                | 329%            |
| poly1           | 25    | 39   | 1.6          | 2                  | 134%     | 4                | 293%            |
| poly2           | 29    | 51   | 1.8          | 3                  | 163%     | 4                | 306%            |
| poly3           | 51    | 81   | 1.6          | 7                  | 231%     | 4                | 327%            |

Table 1 - Summary of results

The Standard Task Graph Set (STG) collection (Kasahara et al. 2007) is a useful source of DAGs, although, apart from being acyclic, many graphs from this collection have an arc/node ratio outside the usual range of a process flowsheet. The shown results are for 5 test graphs from the number of those tested from the STG, for the toy crossflow graph and for 8 flowsheets from practical applications (energy production from fuel cells, multistage evaporation, 4-stage gas compression, specialty chemicals production, and polyolefine production).

## 5. Conclusions

A process flowsheet after decyclification using minimal tearing can be viewed as a task graph which describes the dependencies between the executions of the unit operations. Using this interpretation the flowsheet/task graph can be processed by a scheduling algorithm to obtain a parallel schedule, suitable for optimizing the execution time of a single flowsheet pass on a shared-memory multi-core workstation. By tearing arcs in excess of the minimal tearing required for decyclification it is possible to enhance the speed-up for parallel execution and to limit the depth of the graph.

A practical, simple algorithm is proposed to perform this additional tearing; the proposed algorithm has been tested on a number of randomized graphs and graphs representing actual processes.

The speed-up improvement vary widely, but in general a good increase on 4 cores can be obtained by tearing an additional number of arcs in the same range as the number of arcs to be torn for decyclification.

### 5.1. Future work

The planned future work on this topic includes:

- Implementing more sophisticated scheduling algorithms such as Depth First/Implicit Heuristic Search (DF/IHS) of (Kasahara et al. 1984);
- Optimizing the performance gain when the single-pass (sweep) calculation of the flowsheet is integrated in the algorithms required for the flowsheet resolution;
- Assessing the actual speed-up when execution times differ from those estimated;
- Investigating the effect of increasing the number of cores;
- Integrating some parallelism-improving intelligence in the minimal tearing algorithm.

## References

- P. Aronsson et al., 2002, Proceedings 2<sup>nd</sup> International Modelica Conference, pp. 331-338
- K. V. Camarda et al. 1999, *Comput. Chem. Eng.*, 23, 1063-1073 (1999) doi:10.1016/S0098-1354(99)00271-9
- P. Greppi, 2006, *Proceeding of the International Mediterranean Modelling Multiconference 2006*, pp. 435-440
- Mi-Chang Chang et al., 1988, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design, ICCAD*, pp. 304-307 doi: 10.1109/ICCAD.1988.122516
- H. Kasahara et al. 1984, *Transactions on Computers Vol C-33 (11)* pages 1023-1029 doi:/10.1109/TC.1984.1676376
- H. Kasahara et al. 2007, <http://www.kasahara.elec.waseda.ac.jp/schedule/>
- V. Kulikov et al. 2005, *Chemical Engineering Science* 60 (2005) 2069 – 2083
- Y.-K. Kwok et al., 1999, *ACM Computing Surveys*, Vol. 31, No. 4, December 1999
- OpenMP Architecture Review Board, 2005, *OpenMP Application Program Interface Version 2.5*
- F. E. Sandnes et al. 2005 *Int. J. High Performance Computing and Networking*, Vol. 3, No. 1
- F. F. Wu, 1976, *IEEE Transactions on Circuits and Systems*, vol. CAS-23, No. 12, pp. 706-713.