In HMake, after topological sorting, all nodes are checked. Those with 0 dependencies are added to a list. Then we start building the nodes in this list. As soon as we build a node, we decrement from all of the dependents node.dependenciesSize. If the dependent node.dependenciesSize == 0, i.e. no dependency of the node is left to be built, we add it to the list right before the current iterator.

Now to support the new approach, the following changes are needed.

1) When a file is waiting for another file, its node should mark itself incomplete before returning. So that when the node.update() returns, the build-system does not decrement from its dependents node.dependenciesSize.

2) Whenever a new link is dynamically discovered, we need to do a DFS of the graph from that node to ensure that there is no cycle introduced. We have to check that node that is checked is not checked again.

Suppose this project structure:

A.cpp -> A.hpp

B.cpp -> B.hpp

C.cpp -> B.hpp

A.hpp -> B.hpp

Three source files A.cpp, B.cpp, and C.cpp depend on header-files A.hpp, and B.hpp. Header-file A.hpp depends on B.hpp. We can start compiling A.cpp, B.cpp, and C.cpp as on start these nodes have 0 dependencies each. Suppose we have a 2 thread machine.

1	2	3	4	5	6	7	8	9
A.cpp✓	A.cpp	A.cpp	A.cpp	A.cpp	A.cpp	A.cpp	A.cpp	A.cpp
B.cpp	B.cpp✓	B.cpp	B.cpp	B.cpp	B.cpp	B.cpp	B.cpp	B.cpp
->	A.hpp	A.hpp✓	A.hpp	A.hpp	A.hpp	A.hpp	A.hpp	A.hpp
2 (A.cpp, B.cpp)	->	B.hpp	B.hpp✓	B.hpp	B.hpp	B.hpp	B.hpp	B.hpp
	2 (B.cpp, A.hpp)	->	->	A.hpp✓	A.hpp	A.hpp	A.hpp	A.hpp
		2 (A.hpp, B.hpp)	1 (B.hpp)	B.cpp	B.cpp✓	B.cpp	B.cpp	B.cpp
				->C.cpp	A.cpp	A.cpp✓	A.cpp	A.cpp
				2 (A.hpp, B.cpp)	->C.cpp	C.cpp	C.cpp✓	C.cpp
					2 (B.cpp, A.cpp)	->	->	Exe✓
						2 (A.cpp, C.cpp)	1 (C.cpp)	->
								1 (Exe)

Above is the list of nodes with dependenciesSize == 0 for a sample run. The file that is updated in this step is marked by a checkmark in each step. -> tells us where the list iterator is for the next node addition. While 2 or 1 in the last is the number of active threads. In parenthesis, we specify the files being currently updated by the build-system. The top row specifies the index of the steps that are explained below:

1) Currently we are compiling 2 files, A.cpp and B.cpp. A.cpp discovers A.hpp. As it is not built, it adds it to the list. But before adding, it adds itself to the dependents list of A.hpp and increases its own node.dependenciesSize. It also checks whether adding this new dependency relationship has disturbed the DAG. It also marks itself incomplete, so the build-system won't decrement from its dependent Exe node.dependenciesSize.

2) Similarly, B.cpp discovers B.hpp which is placed in the list.

3) Similarly, A.hpp discovers B.hpp but as it is already added to the list, it is not added.

4) Currently we are compiling only 1 file, B.hpp. B.hpp does not have any dependency, so, its compilation completes. After its node.update returns, build-system decrements from the thrice of its dependents A.hpp, B.cpp, and C.cpp. The thrice of these dependents node.dependenciesSize == 0, so these are added to the list.

5) Now again we are compiling 2 files, A.hpp and B.cpp, whose compilations have now resumed after B.hpp was available. Please notice that the iterator is on C.cpp. Now, A.hpp after compilation completion will add A.cpp. Please notice that this is added before C.cpp. This way the files of the libraries on the top of the DAG are built first so that after their building if they are not to be consumed by their dependents, their resources could be reclaimed. If instead of placing at the iterator spot, we had emplaced the new node at the end, we would not have been able to achieve this.

6) Currently, compiling 2 files B.cpp and A.cpp.

7) Currently compiling 2 files A.cpp and C.cpp.

8) Currently compiling 1 file C.cpp. After this compilation, node.dependenciesSize of Exe is 0 so it is added to the list.

9) Building Exe

We can compile modules similarly. Because of this new approach, I see no issues/limitations introduced for rebuilding, error-handling, or selective-build (building 1 particular target in the DAG of many targets).