MADNESS Algorithms Using the Dataflow Model

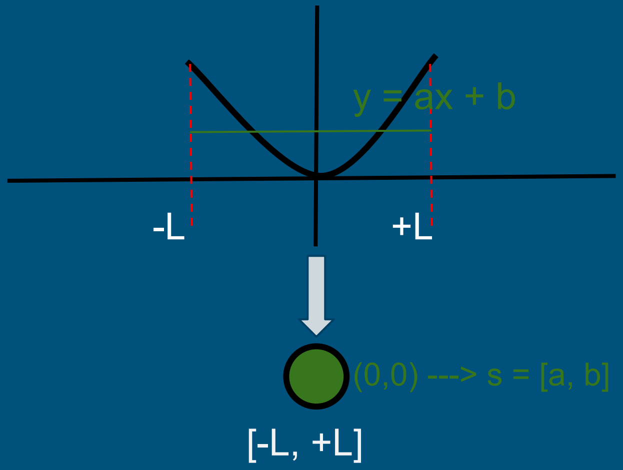
Mohammad Mahdi Javanmard
IACS, Stony Brook University

Introduction to MADNESS

- MADNESS
  - Stands for "<u>**M**</u>ultiresolution <u>**Ad**</u>aptive <u>**N**</u>umerical <u>**E**</u>nvironment for <u>**S**</u>cientific <u>**S**</u>imulation"
  - It can be used as a solution of differential and integral equations in multi-dimensions
  - It has many applications in Quantum Chemistry, Boundary Value Problems, Solid State Physics, Atomic and Molecular Physics in Intense Laser Fields, etc.

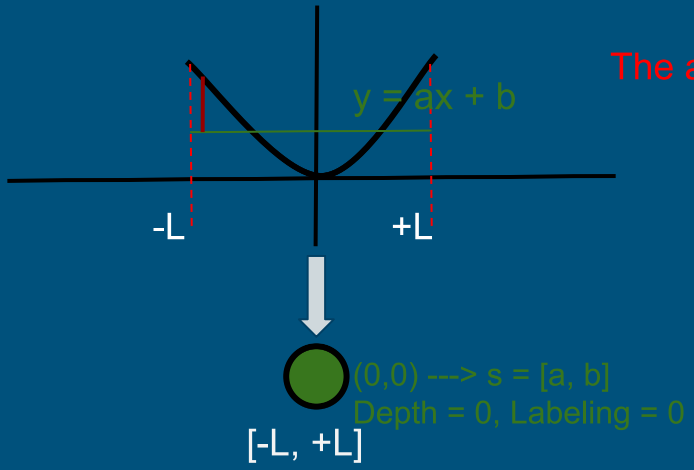
Introduction to MADNESS

- Scientific functions are approximated by a set of simpler functions (for different parts of the function domain):



Introduction to MADNESS

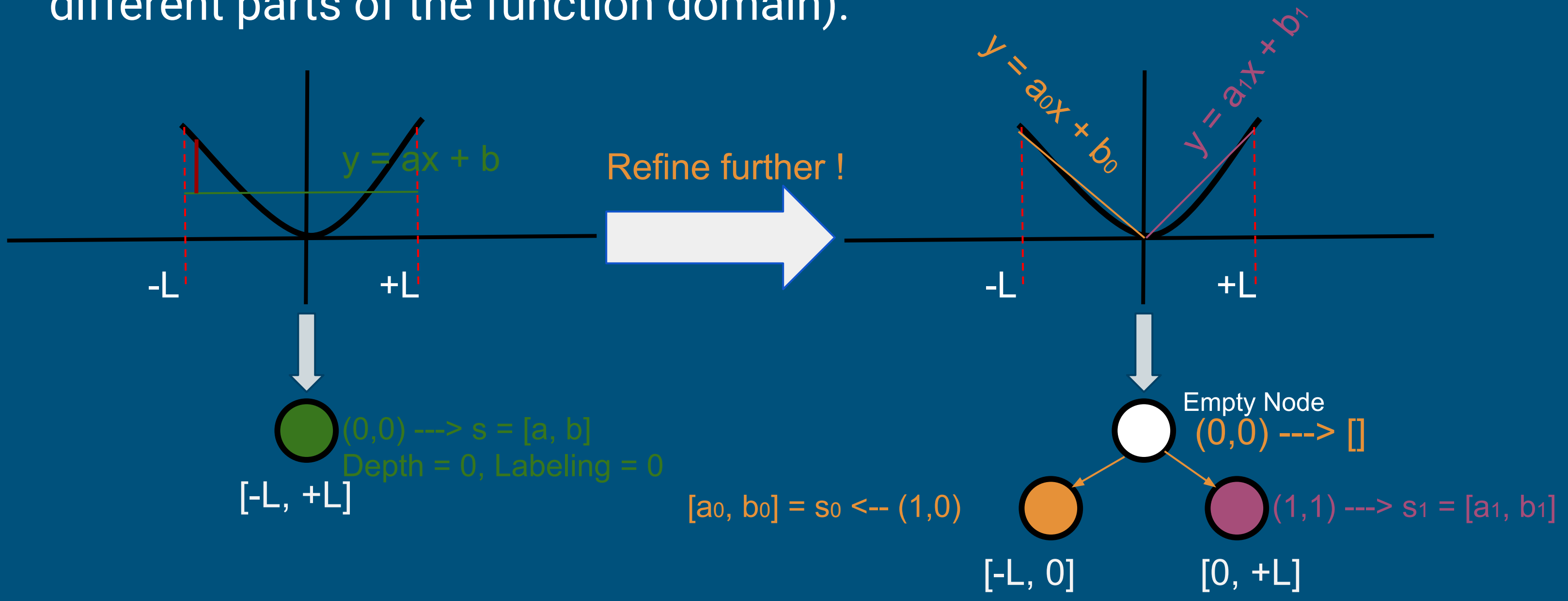
- Scientific functions are approximated by a set of simpler functions (for different parts of the function domain):



The amount of error is not acceptable :(

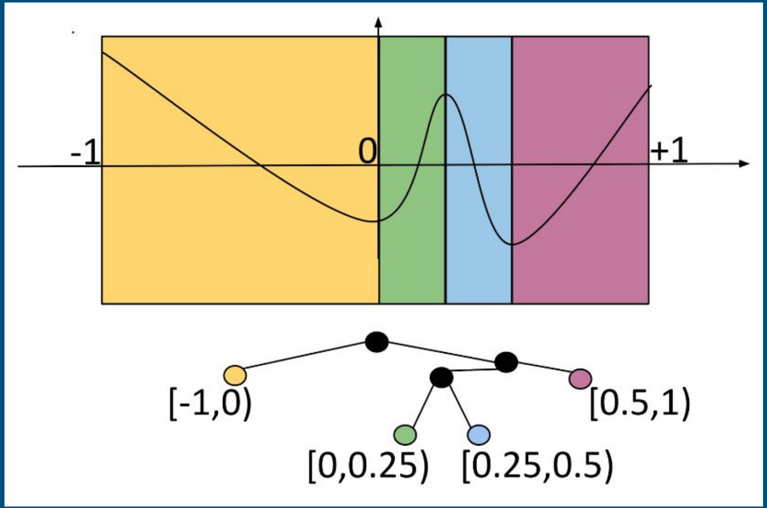
Introduction to MADNESS

- Scientific functions are approximated by a set of simpler functions (for different parts of the function domain):



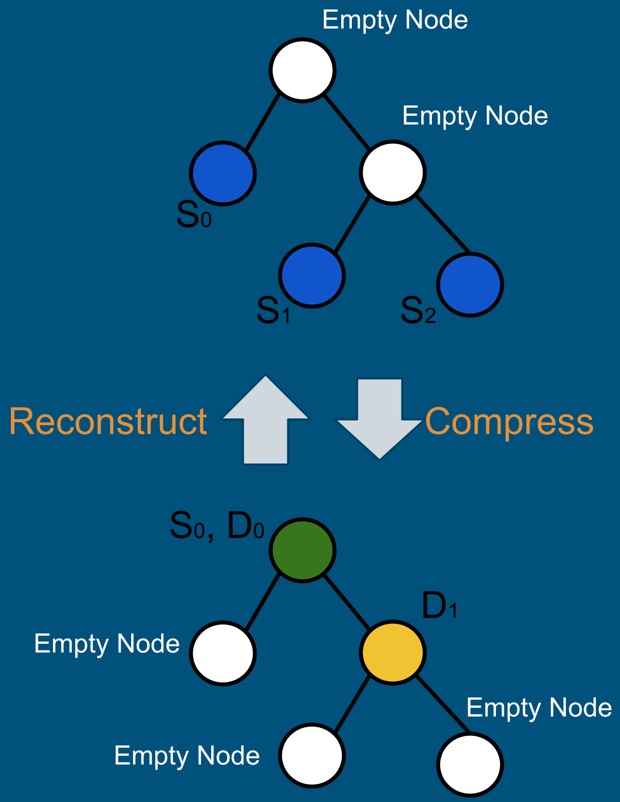
Introduction to MADNESS

- Spatial functions are numerically represented as K-d trees
- MADNESS has several operators (algorithms) which traverses these K-d trees.
- The interesting facts about these trees are:
  - In real applications, the coefficients don’t fit into a memory of a single machine. Hence, having distributed memory paradigm on a cluster of nodes is required !
  - As functions can be complicated at some parts of the domain, the corresponding trees are extremely irregular and not balanced !
  - Yet more interesting, there is no way to predict which parts of the tree are not balanced and irregular !
    - Not that much static optimization techniques applicable.
    - Hence, we need to rely on an intelligent runtime to apply several dynamic optimization techniques.



Introduction to MADNESS

- MADNESS trees can be in either of the following forms:
  - Reconstructed (or refined) form:
    - Data (S vectors) are always on the leaves of the tree.
  - Compressed form:
    - Data (S and D vectors) are in internal nodes of the tree.
    - The name is misleading as the amount/number of data doesn't get decreased !
- Some operators (algorithms) require the operands (inputs) to be in compressed form, others require operands to be in reconstructed form !

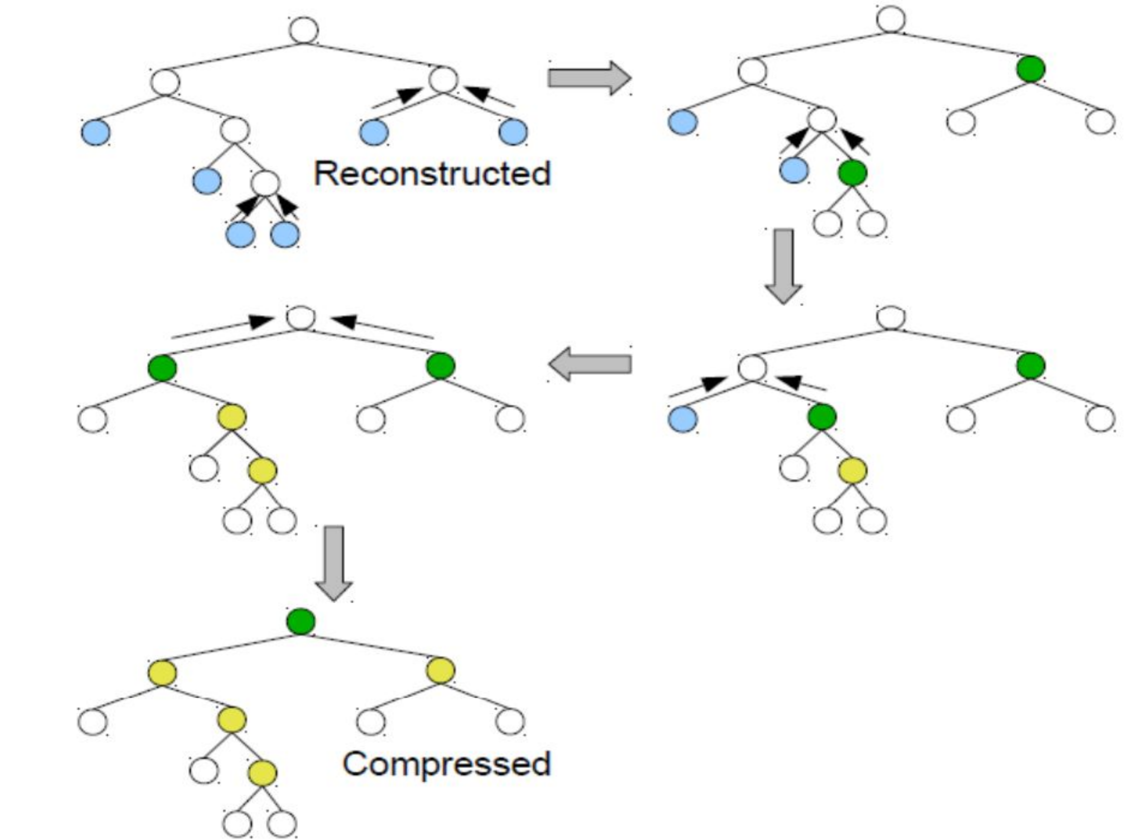


Introduction to MADNESS

- There are several MADNESS algorithms traversing these K-d trees, which can be categorized into:
    - [Strictly] Top-down Traversal
        - Making K-d trees (or Refining K-d trees)
        - Reconstructing the compressed K-d tree
    - [Strictly] Bottom-up Traversal
        - Compressing the tree
    - Either Top-down or Bottom-up:
        - Binary Operators, e.g., sum, multiplication, etc

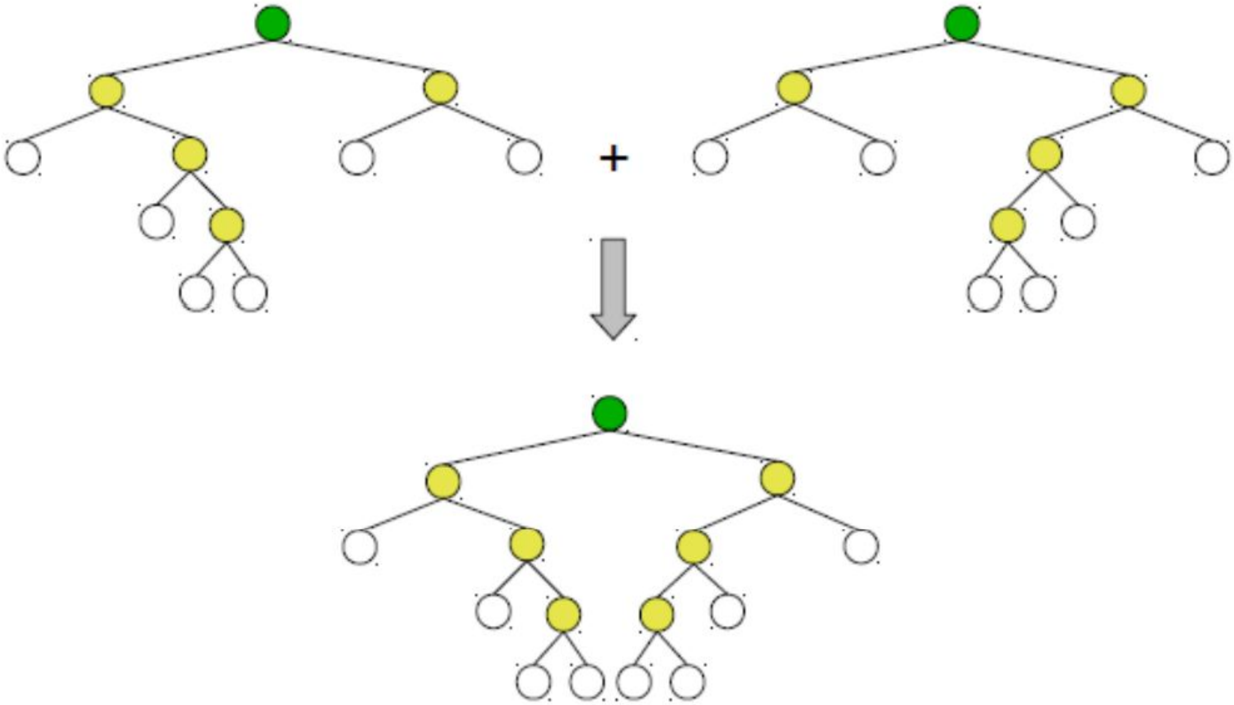
Introduction to MADNESS

- Compress [unary] Operator, as an example of Bottom-up Tree Traversal Algorithm:



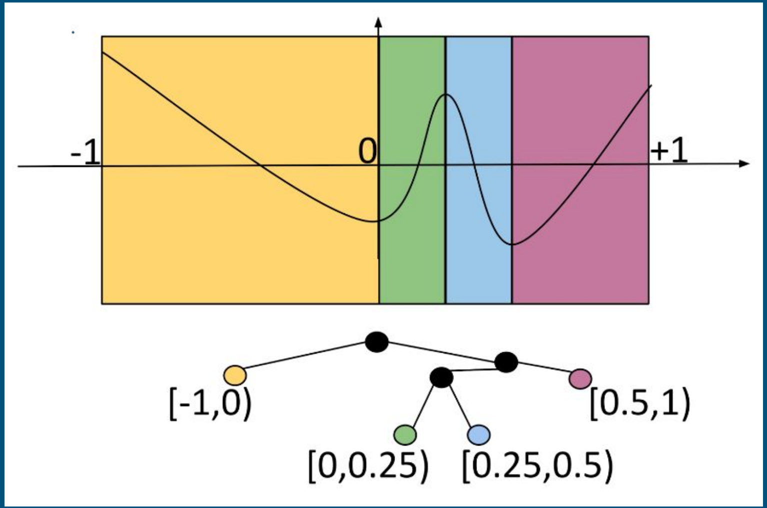
Introduction to MADNESS

- Addition [binary] operator, as an example of Tree Traversal Algorithm:



Introduction to MADNESS

- MADNESS also contains a lightweight task-based runtime which is on top of MPI + Intel TBB. But, (due to nature of fork-join and phase-based paradigms), it has several performance bottlenecks:
  - Global Synchronizations
  - Coarse Grain Parallelism
- A potential solution to resolve these issues? Data-flow model



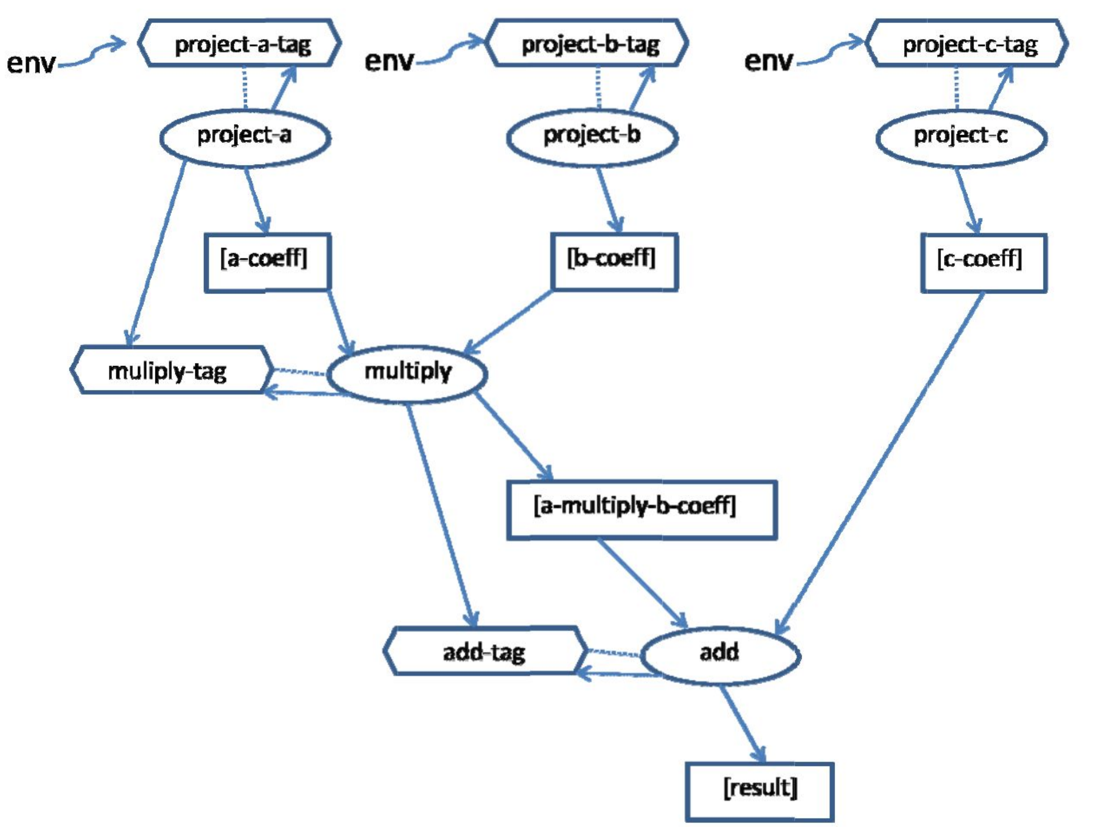
MADNESS+CnC

- As the starting point to implement the MADNESS operators, we looked at the CnC implementation of the MADNESS expression (A*B)+C, where A, B and C are the following functions:

```
double gaussian(double x, double a, double coeff) {
    return coeff*exp(-a*x*x);
}

double test1(double x) {
    static const int N = 100;
    static double a[N], X[N], c[N];
    static bool initialized = false;

    if (!initialized) {
        for (int i=0; i<N; i++) {
            a[i] = 1000*drand48();
            X[i] = drand48();
            c[i] = pow(2*a[i]/M_PI,0.25);
        }
        initialized = true;
    }

    double sum = 0.0;
    for (int i=0; i<N; i++) sum += gaussian(x-X[i], a[i], c[i]);
    return sum;
}
```

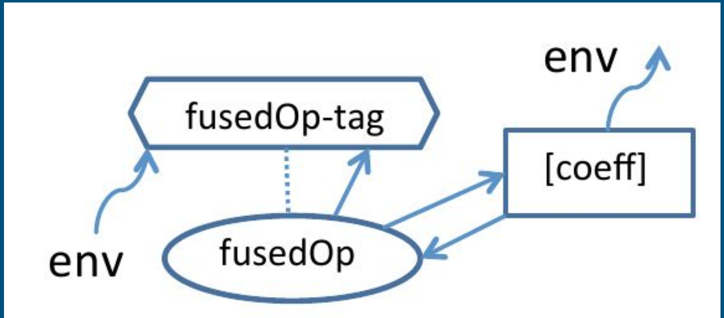
MADNESS+CnC

- A simple (A*B)+C MADNESS computation in CnC:
  - The granularity of the step_collections were node of the trees. I.e., per node of the MADNESS tree, there is an instance of step_collection, for project, add and multiply.



MADNESS+CnC

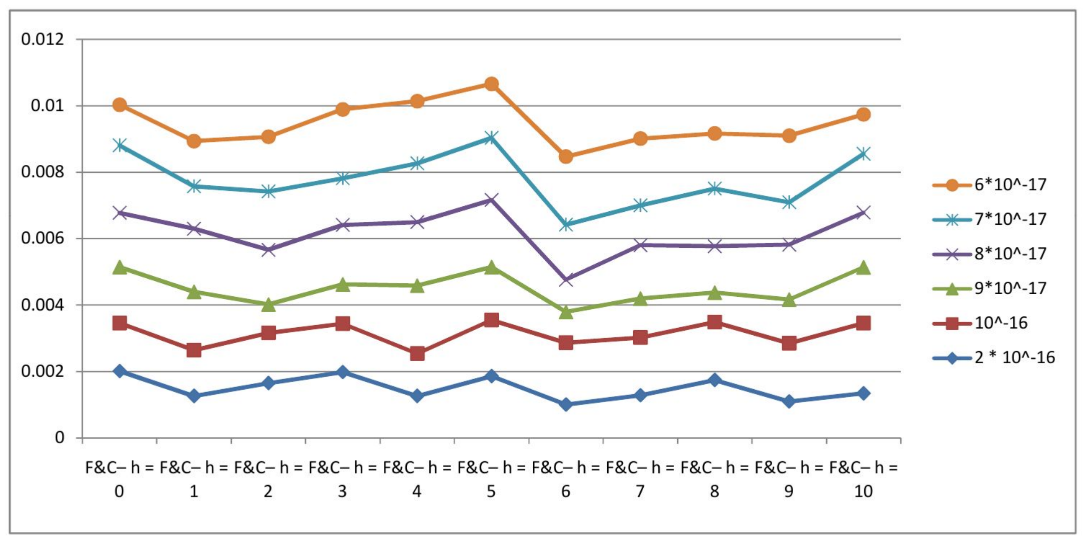
- However, there are two important optimizations which can be easily applied to this computation:
  - Fusing all the operators into one operator (in A*B+C) as all the operators are Top-down:
    - In the new implementation, there is only one step_collection for projecting functions A, B and C, and then, multiplying A and B and then, adding C to get the final result.
  - Coarsening the step_collections to work on small sub-trees instead of working on only one nodes of the MADNESS trees !
- Here is the new CnC computation graph:



MADNESS+CnC

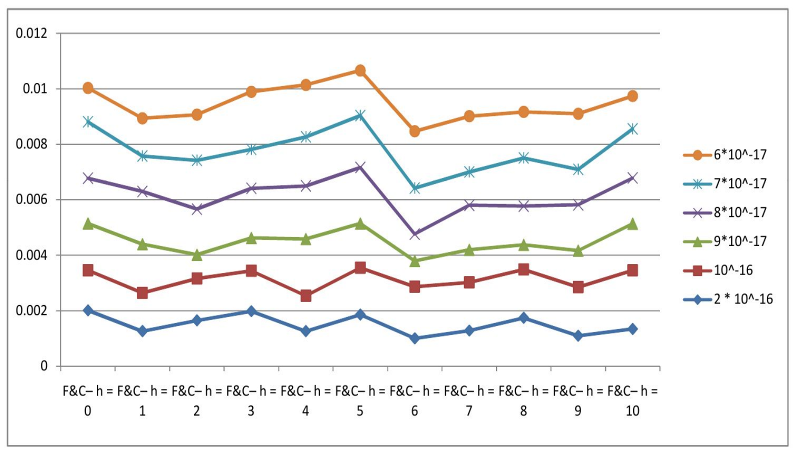
- Following figure shows the result of applying fusing and coarsening:

- X-axis: shows the level of coarsening as a depth of the tree on which the computation step operates.
- Y-Axis: shows the the execution time seconds



MADNESS+CnC

- Lessons learned from the experiment:
  - The optimal height of coarsening the trees step collections are traversing is 6.
  - For the heights less than 6, due to run-time overhead (i.e, generating more instances of step collections), we get worse result.
  - For the heights more than 6, due to having slower step collections (as they traverse bigger subtrees), we get worse result.



MADNESS+CnC

- Future work:
  - As mentioned, it is not possible to fuse all the operators. In other words, we can fuse only operators which are all top-down or all bottom-up. So, we need to automate the analysis of naive CnC computation graph and come up with the fused CnC computation graph.
  - Auto-tuning the execution of step_collections to determine the optimal height for the subtrees to be traversed by the step_collections.