Scaling Out Alternating Direction Isogeometric L2 Projections Solver

Grzegorz Gurgul (AGH)
Bartosz Baliś (AGH)
Marcin Łoś (AGH)
Danuta Szeliga (AGH)
Maciej Paszyński (AGH)

14th U.S. National Congress on Computational Mechanics

July 17-20, 2017, Montreal, Canada

My name is Grzegorz Gurgul. I'm a PHD student at AGH Krakow and a software engineer. I'm going to speak about scaling out iso-geometric ADS solver which we did with Bartosz Baliś, Marcin Los, Danuta Szeliga and Maciej Paszynski.

Agenda

- Background
- Isogeometric L2 projections solver - theory
- Isogeometric L2 projections solver - algorithm
- Isogeometric L2 projections solver - implementation
- Conclusions

Agenda – First I'm going to explain the background of this work. I'm sure most of you are familiar with the algorithm and the theory behind it so I'll only do a brief introduction to those who aren't. What I'd like to focus on in the implementation of the solver because this is what this work is about.

Background

- Isogeometric L2 projections algorithm

Proposed by prof. Victor Calo: L. Gao, V.M. Calo, *Fast Isogeometric Solvers for Explicit Dynamics*, **Computer Methods in Applied Mechanics and Engineering** *(2014)*.

- *Applications to time-dependent problems*

Non-linear flow (Fortran+MPI, parallel): *M. Woźniak, M. Łoś, M. Paszyński, L. Dalcin, V. Calo, Parallel fast isogeometric solvers for explicit dynamics,* **Computing and Informatics** *(2015)*

Tumor growth simulations (C++ sequential): *M. Łoś, M. Paszyński, A. Kłusek, W. Dzwinel, Application of fast isogeometric L2 projection solver for tumor simulations,* **Computer Methods in Applied Mechanics and Engineering** *(2017)*

The initial idea was proposed by Victor Calo back in 2014. There were several implementations proposed which first showed that it produces valid results for some real-life problems.

Background

- Improving performance of time-dependent applications of ADS

Step 1: CUDA implementation:
G. Gurgul, M. Paszyński, W. Dzwinel, Łoś, *GPGPU accelerations of tumor growth simulation using isogeometric L2 - projections solver,* **USACM Conference on Isogeometric Analysis and Meshfree Methods** *(2016)*

Step 2: Object-Oriented shared memory implementation:
G. Gurgul, M. Paszyński, D. Szeliga, Open source JAVA implementation of the parallel multi-thread alternating direction isogeometric L2 projections solver for material science simulations, **Computer Methods in Material Science** *(2017)*

Step 3: Cloud implementation:
G. Gurgul, Bartosz Baliś, Marcin Łoś, D. Szeliga, M. Paszyński, Scaling Out Alternating Direction Isogeometric L2 Projections Solver, **14th U.S. National Congress on Computational Mechanics**

What interests me is how to make use of some properties of that idea to speed up the computations. First I've done it in CUDA environment. Then I made a shared memory CPU parallel implementation which can be seen as intermediate step to move this onto the cluster which this work was about.

Isogeometric L^2 projections

In general: non-stationary problem of the form

$$\partial_t u - \mathcal{L}(u) = f(x,t) \implies \sum_{ij} u_{ij}(B_{ij}, B_{kl})_{L^2} = RHS$$

with some initial state u_0 and boundary conditions

$\mathcal{L}$ – well-posed linear spatial partial differential operator

Discretization:

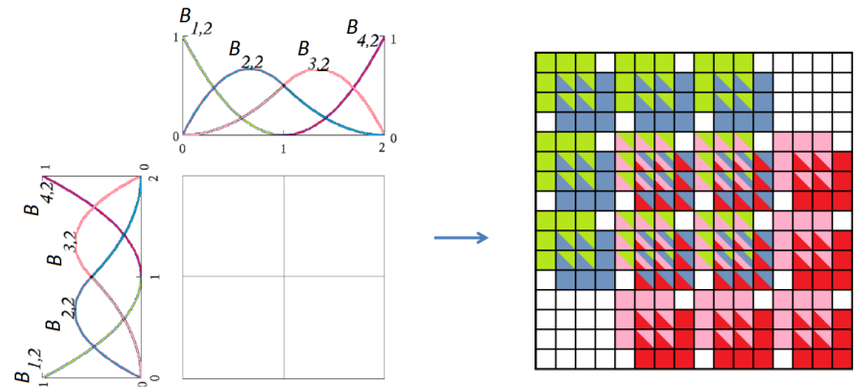
- spatial discretization: isogeometric FEM

  Basis functions: $\phi_1, \ldots, \phi_n$ (tensor product B-splines)

- time discretization with explicit method
- implies isogeometric L^2 projections in every time step

The problem which is solved by this and other implementations is of the following general form. As you can see it's time dependent equation with a well posed linear spatial partial differential operator and some initial state and boundary conditions. We use iso-geometric FEM to discretize the problem in space and explicit time discretization scheme. This implies doing iso-geometric L2 projections in every time step.

L^2 projections – tensor product basis



Isogeometric basis functions:

- 1D B-splines basis $B_1(x), \ldots, B_n(x)$
- higher dimensions: tensor product basis
  $B_{i_1 \cdots i_d}(x_1, \ldots, x_d) \equiv B_{i_1}^{x_1}(x_1) \cdots B_{i_d}^{x_d}(x_d)$

Gram matrix of B-spline basis on 2D domain $\Omega = \Omega_x \times \Omega_y$:

$$\mathcal{M}_{ijkl} = (B_{ij}, B_{kl})_{L^2} = \int_\Omega B_{ij} B_{kl} \, \mathrm{d}\Omega$$

Those iso-geometric projections we need to solve at every time step is just a problem of finding coefficients for the gram matrix in 2D b-spline space. This is costly - the resulting matrix we need to factorize is sparse but not banded - there are some nonzero elements outside of the diagonal which makes gaussian elimination slow.

L^2 projections – tensor product basis

Isogeometric basis functions:

- 1D B-splines basis $B_1(x), \ldots, B_n(x)$
- higher dimensions: tensor product basis
  $B_{i_1 \cdots i_d}(x_1, \ldots, x_d) \equiv B_{i_1}^{x_1}(x_1) \cdots B_{i_d}^{x_d}(x_d)$

Gram matrix of B-spline basis on 2D domain $\Omega = \Omega_x \times \Omega_y$:

$$\mathcal{M}_{ijkl} = (B_{ij}, B_{kl})_{L^2} = \int_\Omega B_{ij} B_{kl} \, \mathrm{d}\Omega$$

$$= \int_\Omega B_i^x(x) B_j^y(y) B_k^x(x) B_l^y(y) \, \mathrm{d}\Omega$$

$$= \int_\Omega (B_i B_k)(x) \, (B_j B_l)(y) \, \mathrm{d}\Omega$$

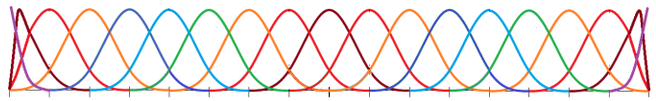
$$= \left(\int_{\Omega_x} B_i B_k \, \mathrm{d}x \right) \left(\int_{\Omega_y} B_j B_l \, \mathrm{d}y \right)$$

$$= \mathcal{M}_{ik}^x \mathcal{M}_{jl}^y$$

$$\mathcal{M} = \mathcal{M}^x \otimes \mathcal{M}^y \quad \text{(Kronecker product)}$$

Luckily there's a property we can make use of. As any higher dimensional spline basis is a tensor product of one dimensional basis we can split each two dimensional coefficient into a product of one coefficient along x and one coefficient along y axis. Then we can group them by x and y which lets us compose two independent integrals along each dimension which can be seen as a Kronecker product of matrices along x and y.

Gram matrix of tensor product basis



B-spline basis functions have **local support** (over $p+1$ elements)
$\mathcal{M}^x$, $\mathcal{M}^y$, ... – banded structure
$\mathcal{M}^x_{ij} = 0 \iff |i-j| > 2p+1$
Exemplary basis functions and matrix for cubics

$$\begin{bmatrix} (B_1,B_1)_{L^2} & (B_1,B_2)_{L^2} & (B_1,B_3)_{L^2} & (B_1,B_4)_{L^2} & 0 & 0 & \cdots & 0 \\ (B_2,B_1)_{L^2} & (B_2,B_2)_{L^2} & (B_2,B_3)_{L^2} & (B_2,B_4)_{L^2} & (B_2,B_5)_{L^2} & 0 & \cdots & 0 \\ (B_3,B_1)_{L^2} & (B_3,B_2)_{L^2} & (B_3,B_3)_{L^2} & (B_3,B_4)_{L^2} & (B_3,B_5)_{L^2} & (B_3,B_6)_{L^2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & (B_n,B_{n-3})_{L^2} & (B_n,B_{n-2})_{L^2} & (B_n,B_{n-1})_{L^2} & (B_n,B_n)_{L^2} & \end{bmatrix}$$

Each of those matrices are not only sparse but also banded. This is because one dimensional b-splines have local support and a product of two of them is not zero only if they lie next to each other. To be precise each matrix band is going to be 2p + 1 wide where p is the order of the spline used.

Alternating Direction Solver – 2D

Two steps – solving systems with **A** and **B** in different *directions*

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & 0 \\ A_{21} & A_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} y_{11} & y_{21} & \cdots & y_{m1} \\ y_{12} & y_{22} & \cdots & y_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ y_{1n} & y_{2n} & \cdots & y_{mn} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{21} & \cdots & b_{m1} \\ b_{12} & b_{22} & \cdots & b_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ b_{1n} & b_{2n} & \cdots & b_{mn} \end{bmatrix}$$

$$\begin{bmatrix} B_{11} & B_{12} & \cdots & 0 \\ B_{21} & B_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_{mm} \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & \cdots & x_{2n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mn} \end{bmatrix}$$

Two one dimensional problems with multiple RHS:

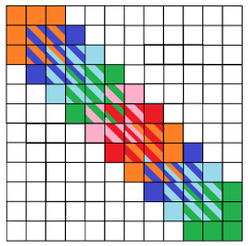
- $n \times n$ with m right hand sides $\rightarrow O(n * m) = O(N)$
- $m \times m$ with n right hand sides $\rightarrow O(m * n) = O(N)$

Linear computational cost $O(N)$ as opposed to $O(N^{1.5})$ or $O(N^2)$

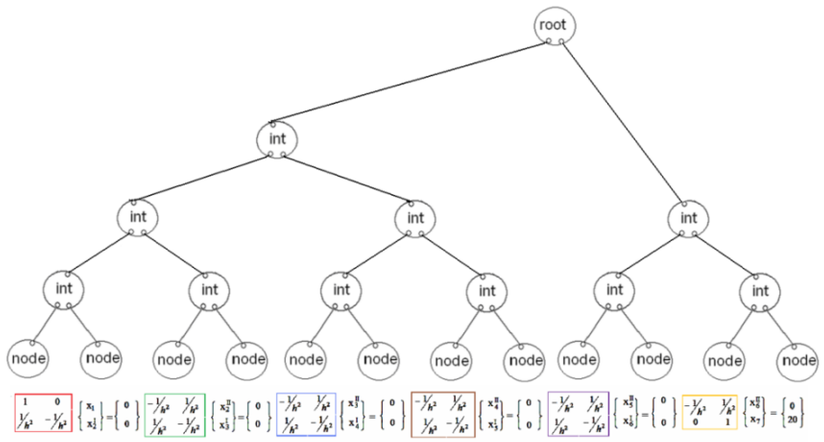
After some transformations on the block matrix of Kronecker product we can actually present the problem in the following form. We have to solve a partial problem in one dimension and use that information to find the solution for the other dimension and to the initial problem. So rather than having to solve one problem with a matrix difficult to factorize we now have to solve two problems with matrices easy to factorize at a cost of having multiple right hand sides. This makes the cost linear as opposed to N^1.5 or N^2 in 3D.

Algorithm - data structure

The backing data structure is a tree. Each of its nodes holds coefficients of a simple linear equation being a portion of the original system.



Partition of the problem matrix into sub-matrices



Graph of matrices the solver will operate on

We can use the idea of multi-frontal solver to find the solution for each of those two problems.

This is done by splitting each set of linear equations into subsets which are bound to certain nodes of a tree.

The tree itself can be seen both as a data structure and a scaffolding the algorithm operates on.

By applying a series of operations on each level moving up and down the tree we can find solutions to all problems.

Algorithm - set of operations

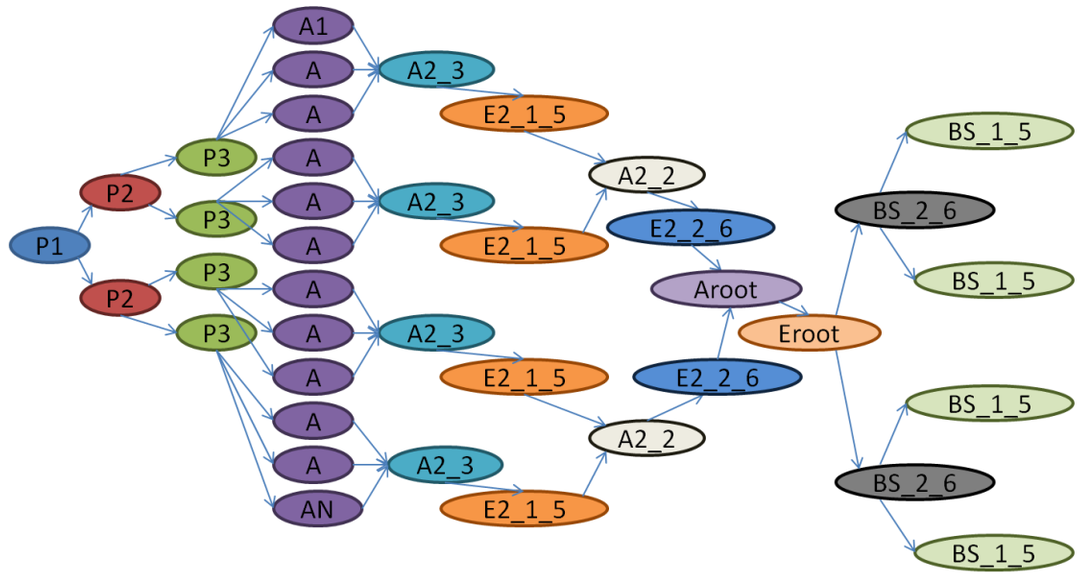
We can identify a set of basic tasks applicable on any vertex. We call them **productions**. They can:

- branch a vertex into two $\{P1, P2\}$ or three $\{P3\}$ child vertices
- initialize a vertex with particular coefficients $\{A1, A, AN\}$
- merge two vertices and eliminate unknowns - $\{A2\_3, E1\_2\_5, A2\_2, E2\_2\_6, Aroot, Eroot\}$
- backward substitute parts of solution - $\{BS\_2\_6, BS\_1\_5\}$

Those operations applied on vertices are called productions. There are operations which create a tree, initialize coefficients at the nodes, do factorization and backwards substitute to find a solution.

Algorithm - flow

To obtain a solution for a 1-dimensional problem with 12 elements it is enough to execute the following productions, set by set, going from left to right, on respective vertices.



Here is the illustration of solving the basic problem with 12x12 matrix. We first create the tree, then initialize it, then factorize and backward substitute. What is important is that any two operation in one column are independent from each other and thus can be done in parallel. Of course you may say that it is only the bottom of the tree which can benefit out of it but the truth is we care only about that bottom because it is where the most demanding operations take place. The rest is only multiplying and flipping rows which is negligibly fast.

Implementation

CUDA (GPU):

- extremely fast for problems which *fit in on-board memory*
- verbose and fragile
- hardware dependent

Shared memory (Multicore CPU)

- fast for problems which fit in RAM (37 million elements require 16GB of RAM)
- portable, easy to understand and adapt
- scales up only

In memory grid (Cloud)

- can solve problems of any size
- slower for relatively small problems
- *scales out*

So whole point now is to do this in an environment which can do a lot of things in parallel.

The GPU based implementation I mentioned earlier proved to be lightening fast but also very limited as it could solve only small problems. To make things worse it was difficult to change, maintain and required specific hardware.

On the other hand a shared memory implementation was very flexible but still could be scaled only up. This could solve a problem up to 37 million elements in mesh.
It can be seen as an intermediate step to move onto cloud.

The implementation I want to talk about today is made on top of in memory data grid. It can scale out and run complicated problems even on low-end nodes.

Implementation - performance considerations

Distributed environment implies *heavy network traffic* and increased memory consumption due to *extensive serialization*. This has to be reduced to the absolute minimum to let solver run bigger problems.

The following measures have been used:

- split tree into set of subtrees of a given height and store them on a single node
- use localized map-reduce to extract columns from the solution rows
- reuse same operation on multiple vertices
- run subsequent operations applied on same vertex in one invocation
- schedule tasks in batches
- use near-caches
- keep solution in deserialized form

In practice there are many issues that need to be addressed not only to make this work fast but to make it work at all. Two things which we have to consider is network overhead and memory consumption during serialization.

Several tricks had to be used to lower those impediments to the absolute minimum. Without them adding new node would degrade performance rather than improving it.

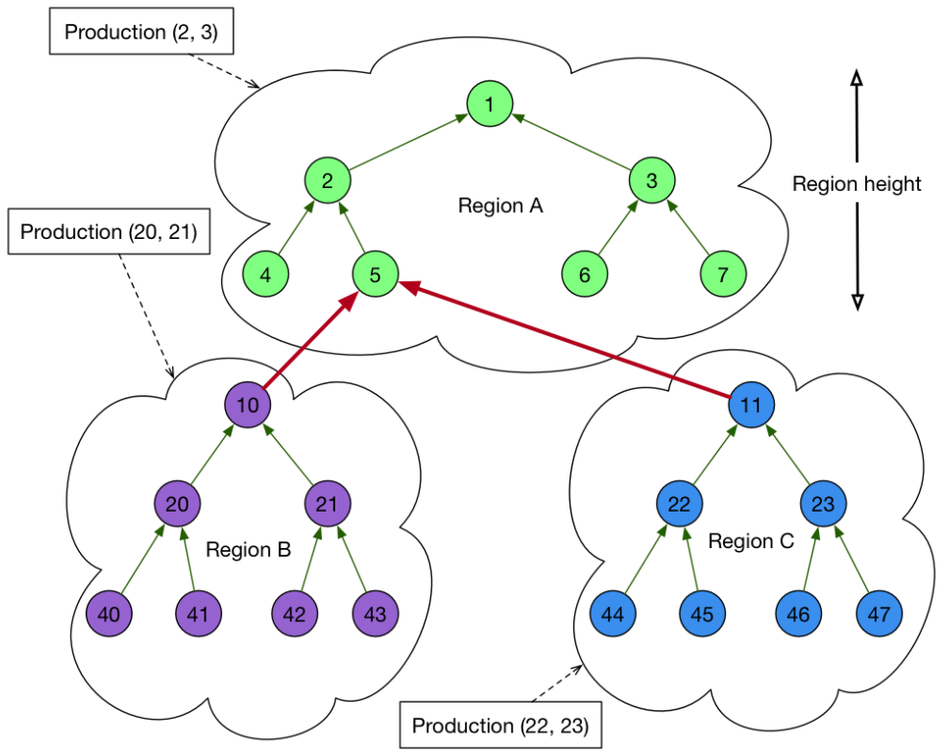
For instance we want to keep relevant the data at the node the computations are scheduled on. This is why the tree is partitioned into subtrees stored on particular nodes.

We also want to limit the number of invocations. We do this by reusing same operation on multiple vertices as well as batching those operations if they are run one after another on the same set of vertices.

We also want to use near-caches for read-only data.

We use localized map-reduce algorithm to extract columns from solution rows. Doing this differently would hinder performance by a magnitude and introduce a bottleneck limit to the maximum problem size.

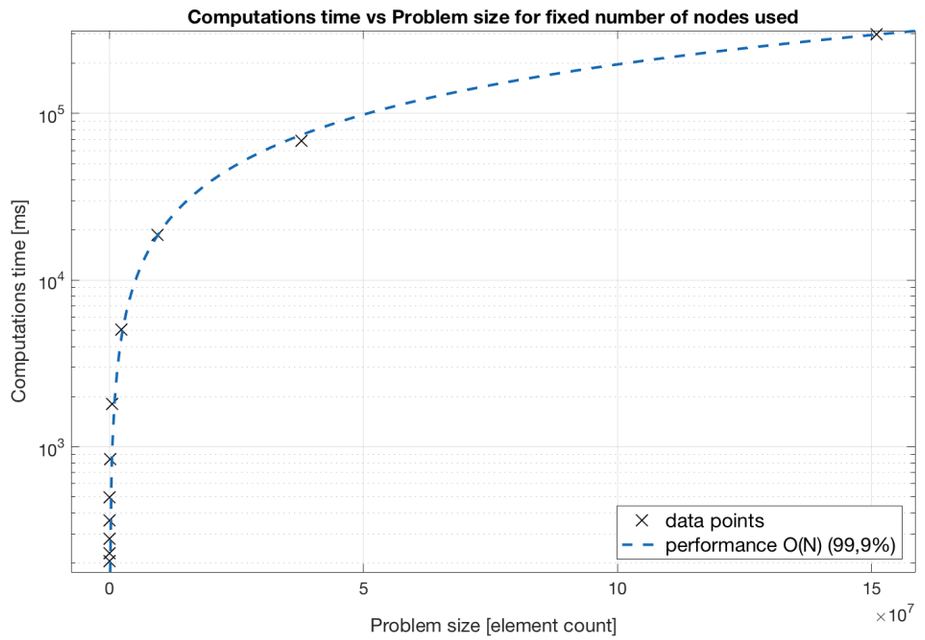
Implementation - partition tree



This is the illustration of how tree partitioning works. A tree is split into regions - in this case a region has height 3 so we have 3 regions for a tree of height 6. As productions launched require data only from the node they are launched on and its direct children any operations done on nodes 2,3,20,21,22,23 are going to be fast. Operations on node 5 is going to be slower for the very same reason.

The higher the region the fewer network interactions there will be. On the other hand big regions limit the parallelism to a single node and may be inefficient. It's a tradeoff. For problems represented by trees of height 12 the best region height was 4 to 6 depending on node count.

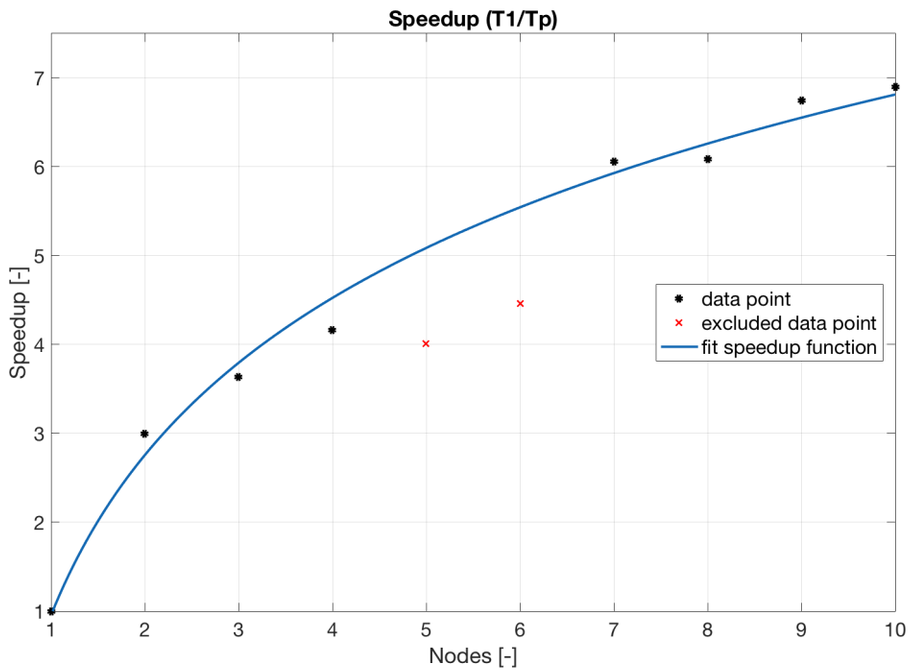
Implementation - time complexity - O(N)



* Run on a cluster of 11 machines - Westmere (Nehalem-C 2.7GHz) CPU with 4 cores / 8GB RAM each

Here is the time complexity of the IMDG implementation. It’s perfectly linear for a fixed number of nodes.

Implementation - speedup

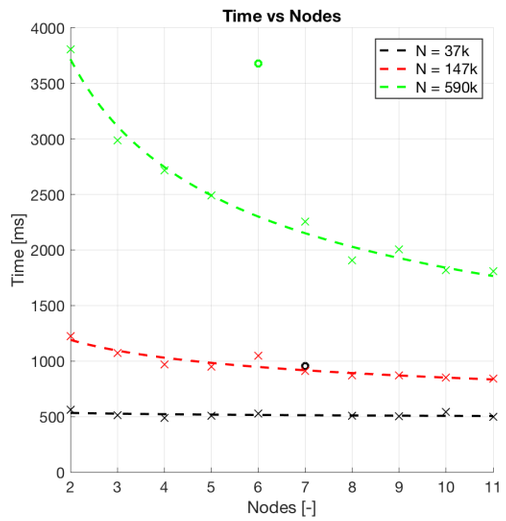


* Run for a fixed problem size. Each node has Westmere (Nehalem-C 2.7GHz) CPU with 4 cores / 8GB RAM each

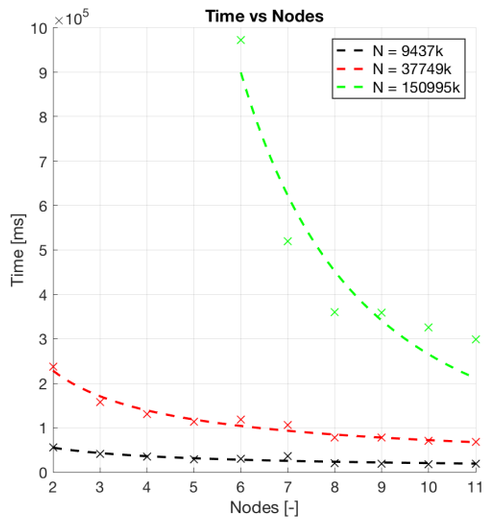
Here is the speedup diagram. It's non-linear as it has been drawn for rather small problem size for which applying additional nodes brings smaller and smaller benefit. On the other hand we could not draw it for a big problem size because the bigger the problem the more memory it requires so it cannot be computed on small number of nodes. If we look at the results between 2 and 7 nodes they are indeed linear as the problem size is big enough for those nodes to distribute enough load between them.

The huge gain from using two nodes as opposed to only one is because not only this second node does half of computations but also let the first node to do its part more efficiently - normally it would have to do a lot of garbage collections in a meantime wasting CPU cycles.

Implementation - scaling out



Small problems ($2m$ elem.)



Larger problems ($2m$ to $150m$ elem.)

* Each node has Westmere (Nehalem-C 2.7GHz) CPU with 4 cores / 8GB RAM each

Here is how it looks like for different problem sizes - looking at the time. For small problem sizes there is no gain at all. The bigger the problem the more gain we get out of applying additional nodes. Computing a solution for a problem with 150M elements (12k elements in one direction) take around 5 minutes. The biggest problem I could run on some high-end 12 core 16gigs of RAM machine took around that to compute 4 times smaller problem. On the other hand it took considerably less to find the solutions for smaller problems (up to 1536^2 elements)

Conclusions

- Distributed memory implementation is slower than shared memory one for small problem sizes which fit into physical memory of a single machine
- It can solve any problem by adding additional nodes (4 - 6144, 6 - 12288, 12 - 24576)
- Given there is a shared memory machine with sufficient physical memory, IMDG implementation outperforms it starting from a certain problem size
- This implementation can be made significantly faster by pushing some logic into worker nodes

The conclusions are that if the problem fits into the physical memory and we have a dozen of cores in CPU it's best to use shared memory implementation which will be much faster and easier to run.

But if the problem doesn't fit into the physical memory of a single machine and rather than having a high-end PC we have multiple low end ones we can go with this approach which scales by adding additional nodes. This way using 4 nodes we can compute a problem of 6144 elements in one dimension, using 6 12288, using 12 24576 and so on.

Given we have a node with enough physical memory to run any problem starting from a certain problem size it's still going to be slower than running it on cluster implementation due to lower computing power.

This cluster implementation could be made much faster by pushing a logic of the computations onto the worker nodes themselves rather than orchestrating computations from a client node which implies additional traffic and delays.

Thank you!

Our research is supported from Dean's grant from Faculty of Computer Science, Electronics and Telecommunication, AGH University, Krakow, Poland

This work has been supported from a Dean's grant of department of Computer Science, Electronics and Telecommunication of AGH