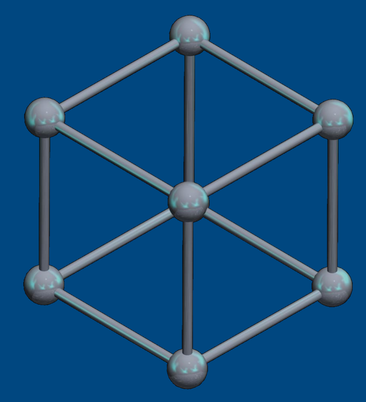
Solving Communication-Intensive Problems Efficiently Using On-Chip Mesh Interconnection Networks

A design study on a novel data flow processor architecture

Sebastian Klaassen
Department of Computer Science, University of Vienna, http://cs.univie.ac.at/



Introduction

Traditional concurrent software focuses on adapting established sequential information processing practices towards increasingly parallel execution, but communication between individual nodes is an unavoidable bottleneck.
Highly parallel data flow architectures, however, see communication as the basis for concurrent performance. They have evolved from a theoretical concept for parallel hardware to supercomputers like the Connection Machine.
This poster documents the creation of a new dataflow processor, named the Mesh Processing Unit (MPU), by (1) finding a versatile topology, (2) simulating hardware in C++ code, (3) creating a domain specific language for formulating algorithms, (4) creating a rich development environment for debugging and analyzing MPU code execution and (5) assessing usability and potential application areas for the architecture.
In contrast to existing data flow architectures (Wavefront Array Processors, Systolic Arrays, Network on Chip Architectures, …) the MPU maps data flow directly to the underlying algorithmic topology. By placing a 2D or (virtual) 3D mesh of simple processor cores on a chip, each connected to its neighbor's memory banks and driven by a global system clock, numerous algorithms can be efficiently computed in parallel.
This poster is grouped into 2D topology and algorithms, 3D topology and algorithms, simulation software and conclusion and future work.

Mesh topologies are based on regular tilings of the plane:

2D Cartesian Mesh

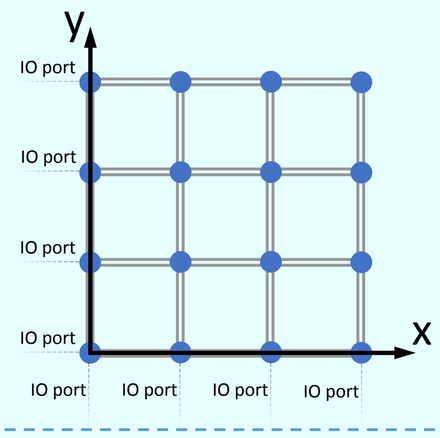
The simplest topology for a 2 dimensional network of processors is the 2D cartesian mesh. From each processor data can flow in four directions. This structured grid maps naturally to a large area of problem domains, like linear algebra or finite element methods.

Data flows in through IO ports, gets processed in the mesh and flows back out of these IO ports.

Data can flow infinitely far in logical x- and y direction, because infinitely many nodes are mapped onto the physical grid using torus mapping.

Information about what operations to perform are passed along with the data (see 'Programming the MPU').

Each processor has a memory queue that processes data on a first-come first-serve basis.



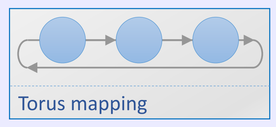
Logical 3D Cartesian Mesh Mapped Onto 2D Hexagonal Grid

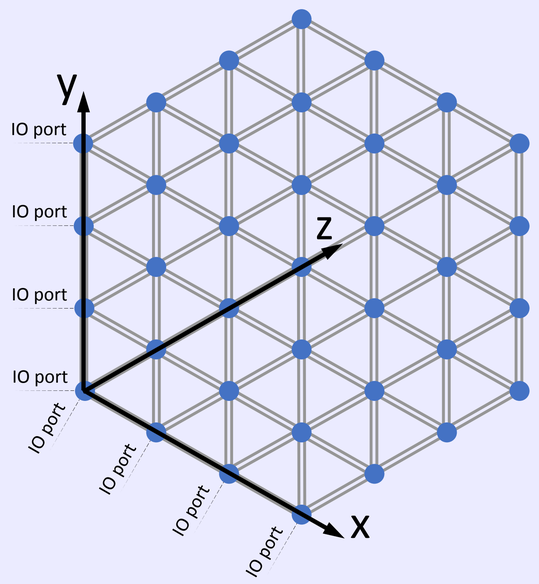
3 dimensional algorithms can be implemented by creating a logical 3D mesh over a physical 2D grid. The isometric projection of a 3D cartesian grid yields a hexagonal layout in the plane.

Two logical processors p and q who's coordinates fulfill the equations $p_x + p_z = q_x + q_z$ and $p_y + p_z = q_y + q_z$ are mapped to the same physical processor.

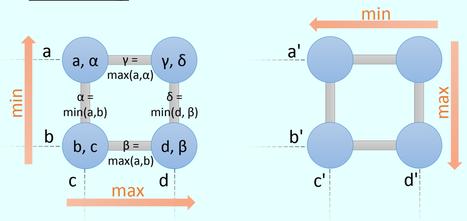
The advantage of such a projection over simpler orthographic projections is that data traveling along either axis traverses physically separate processors, therefore enabling parallelism.

Again the logical mesh is an infinitely large mesh torus-mapped onto physical cores.





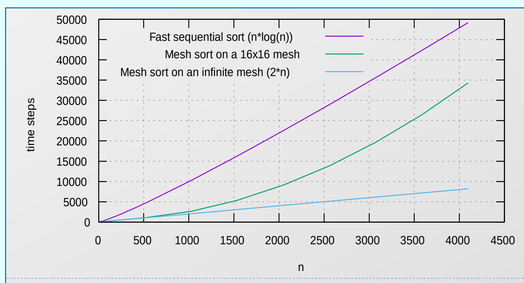
Sorting



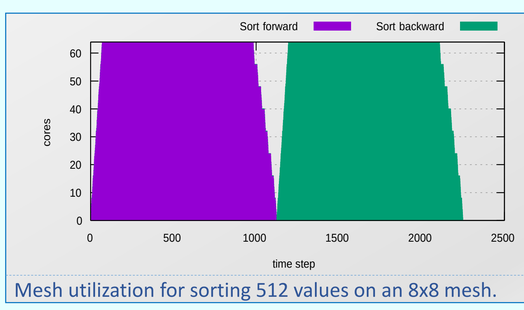
1) Small values float up
   Large values float right

2) Small values float left,
   Large values float down

The best sequential sorting algorithms are proven to have an order of $n \log(n)$. The implemented mesh sort algorithm is a bitonic sort based algorithm. See above for an illustration on how to sort a set (a, b, c, d) into (a', b', c', d'). It has a sequential order of $2\ n^2$ (total number of time steps performed by all cores), but a parallel order of $2\ n$ (number of observed time steps on an infinitely large mesh). On a finite mesh execution time lies somewhere in-between (see runtime plot below).

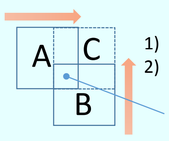


Runtime comparison of mesh sort on a 16x16 mesh with a fast sequential algorithm and mesh sort on an infinite mesh.



Mesh utilization for sorting 512 values on an 8x8 mesh.

Dense Matrix Multiplication

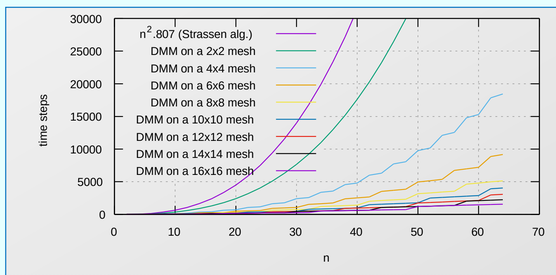


1) Push matrices into each other
2) Multiply-add inside overlapping area
   $c_{ij} = c_{ij} + a_{ij} * b_{ij}$

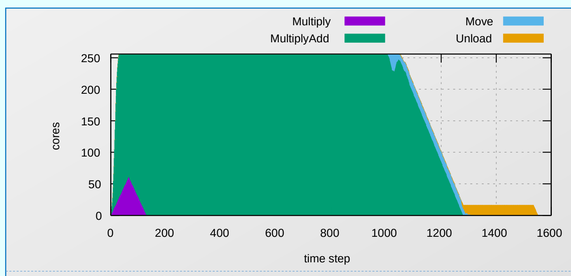
Basic dense matrix multiplication (DMM) consists of n^3 operations. Optimized sequential algorithms achieve around $n^{2.807}$ (Strassen alg.). The biggest challenge of parallel DMM lies in efficiently distributing workload.

MPU matrix multiplication utilizes all cores during computation (see *'Multiply'* and *'Multiply & Add'* in the utilization plot below).

On a mesh with mxm cores, the computation requires $O(\frac{n^3}{m^2})$ operations, but getting the computed matrix out through the IO ports adds an $O(\frac{n^2}{m})$ overhead (see *'Move'* and *'Unload'* in the utilization plot below).



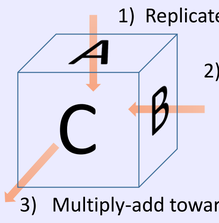
Runtime comparison of matrix multiplication on different mesh sizes with the Strassen algorithm. Ripples in curves show that DMM is most efficient for matrices where n is a multiple of the mesh size.



Mesh utilization for multiplying two 64x64 matrices on a 16x16 mesh. Movement steps push the final matrix toward the IO ports for unloading.

3D Dense Matrix Multiplication

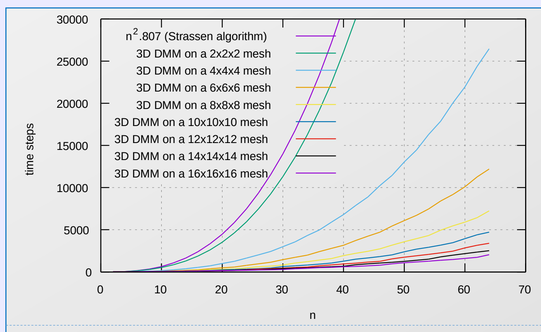
1) Replicate A downwards



2) Replicate B sideways
3) Multiply-add towards the front

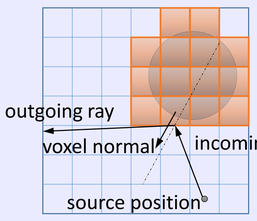
To test the implementation of a highly **structured** algorithm in 3D domain, matrix multiplication has also been implemented for the 3D cartesian mesh.

Here two matrices are loaded onto two sides of the cube that is spanned out by the 3D mesh and replicated throught the whole mesh in the direction of their respective normals. Once every logical node within the cube holds one value of each matrix, those values are locally multiplied and summed up towards the third side of the cube.



Runtime comparison of 3D matrix multiplication on different mesh sizes with the Strassen algorithm.
3D DMM is slower than 2D DMM while requiring more processors. It might however still be used in cases where input matrices are already in orthogonal locations after a previous computation.

Ray Tracing

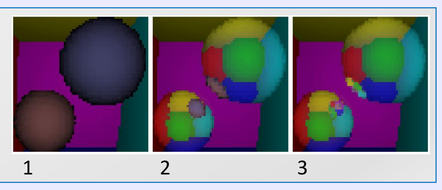


To test the implementation of an **unstructured** algorithm in 3D domain, ray tracing of recursive specular reflections (mirror-like reflections without diffusion) has been implemented.

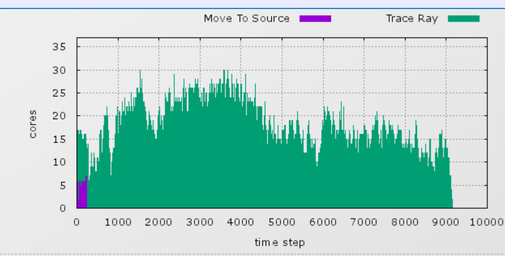
The scene is divided into voxels and rays are casted by marching voxels in ray direction.

Before casting rays from the camera position into the scene, ray parameters are fed through an IO port and travel to the source position (see *'Move To Source'* in the utilization plot below). After reaching a preset number of reflections (light bounces) the ray is unloaded from the mesh through the nearest IO port.

Resulting images



# of reflections: 1 2 3



Mesh utilization for casting 32x32 rays into a scene of 100x100x100 voxels on a 4x4x4 mesh.
A high utilization depends on an even distribution of rays among voxels. Effective utilization averages around 50%.

Programing the MPU

All plots on this poster measure time in '**time step**'-units. Within one time step each core can execute a single **task**.

A task consists of (1) retrieving inputs from connected cores, (2) performing operations on them, and (3) returning the results to one or more connected cores.

We have created a simple Python based domain specific language to formulate such tasks (see below).

```
def ES_Sort(pos, south, west):
    east = Output(
        value = min(west, south),
        task = ES_Sort if pos[0] + 1 < N else ES_FlipX
    )
    north = Output(
        value = max(west, south),
        task = ES_Sort if pos[1] + 1 < N else ES_FlipY
    )
    return RunResult(north = north, east = east)
```

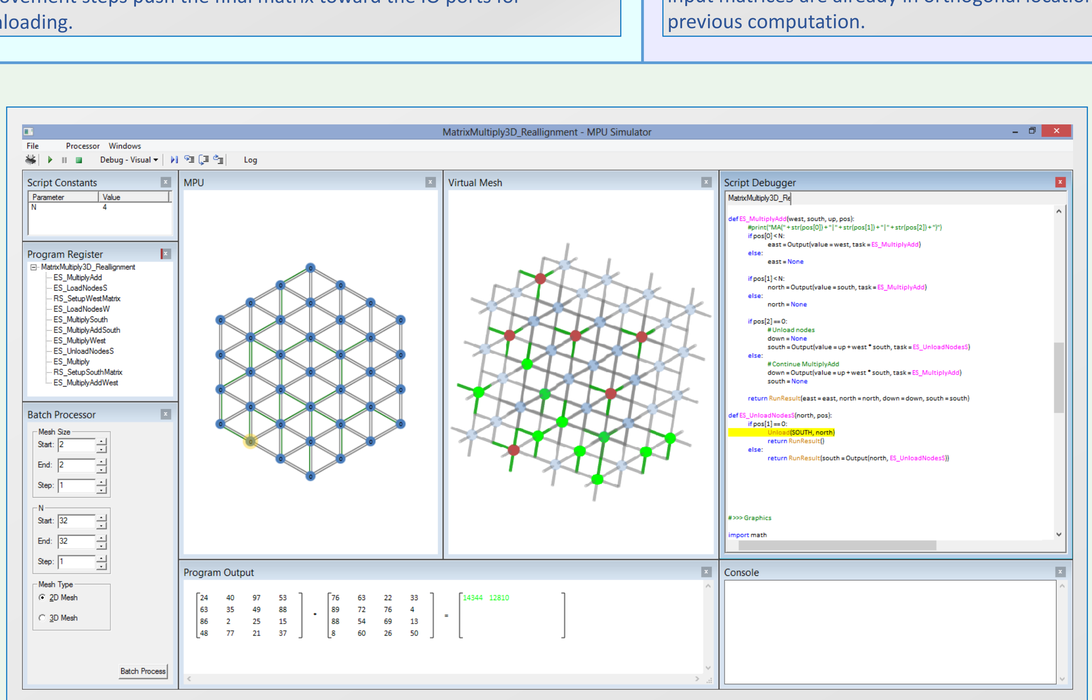
The forward sorting task from mesh sort:
South- and west neighbors provide input to this task. 'pos' holds the current logical mesh coordinates. Each output contains a data value and a pointer to the next task. Here the next task is again a sorting task, unless all data has been fully traversed, in which case a flip-task prepares for subsequent backward sorting.

Simulation Environment

To execute MPU code, we are simulating MPU topology and anatomy of individual cores using a C++ library, called the MPU runtime.

The development of a rich programing environment to control and debug program flow and visualize both logical mesh and physical processor grid has proven essential throughout our research.

The simulation environment is a separate software component from the MPU runtime.



The MPU development environment showing the execution of a 3D matrix multiplication of two 4x4 matrices on a 4x4x4 mesh. Code is currently being halted during the execution of an 'unload' task by the processor at logical position (1, 0, 0).

- The '**MPU**' window shows the physical grid highlighting active connections.
- The '**Virtual Mesh**' window shows a 3D rendering of the logical mesh, with node colors indicating processor utilization.
- The '**Script Debugger**' window shows MPU code, supporting line-by-line debugging and variable peaking.
- The '**Program Output**' window renders computed values in a customizable fashion. Currently rendering the two input matrices and values of the output matrix computed so far.
- The '**Program Register**' gives an overview of all defined tasks (MPU subroutines) that are stored in program memory of all processors. The 'Batch Processor' window was used to create all the graphs on this poster.

Conclusions and Future Work

- Sorting and DMM achieve full mesh utilization throughout most of the computation. Data move operations to- and from IO ports reduce load balancing through data collisions, causing congestion.
- Unstructured algorithms like ray tracing have a significantly lower average load balance because of unpredictable data flow.

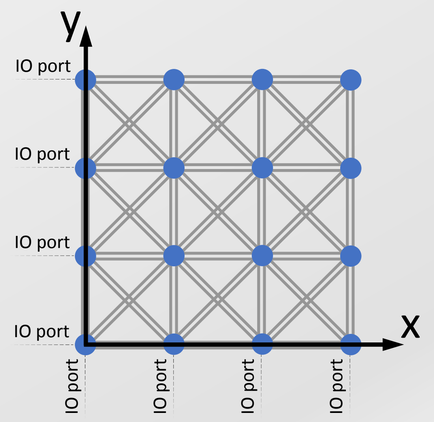
- Topologies with more connections should help reduce congestion by introducing congestion avoidance for data movement operations (see below)
- The IO bottleneck constitutes a significant part of the total runtime. More complex algorithms will reduce the relative impact of IO by keeping data in the mesh longer.
- Another bottleneck is realignment of input sets to the location and shape expected by the algorithm. More advanced compilers should support more flexible input alignment and optimize for minimum realignment between subprograms.

- Important future work will be to develop a prototype MPU with a sufficiently large mesh that supports at least a subset of Basic Linear Algebra Subprograms (BLAS), so the usefulness of the MPU as a coprocessor can be studied.

2D Cartesian Mesh with Additional Diagonal Connections

Previous work suggested that a 2 dimensional topology with torus mapping is sufficient for most problem domains.

Additional diagonal connections are added to allow more sophisticated congestion avoidance strategies for data transfer operations. The extra connections also raise area-efficiency.



universität wien