Supercomputing Frontiers and Innovations

2014, Vol. 1, No. 3

Scope

- *Parallel and distributed computing technologies.* Parallel programming languages and libraries. Methods, algorithms and tools for analysis, debugging and optimization of supercomputing applications. Co-design concepts.
- *Next-generation supercomputer architectures.* Advanced supercomputer architectures. Reconfigurable and hybrid supercomputing systems. Accelerator based supercomputing systems. Modeling and development of new high-performance computing hardware.
- *Supercomputing Applications.* Solving of computationally intensive problems using supercomputers in computational mathematics, computational physics, chemistry, hydro-gas-dynamics and heat transfer, nonlinear transient problems in mechanics, bioinformatics, engineering, nanotechnology, climate, ecology, cryptography, and other prospective areas.
- *Visualization.* Parallel methods and algorithms of visualizing of computational experiments.
- *Management, administration and monitoring of supercomputing systems.* Methods, algorithms and tools of management, technical support and monitoring of highly parallel supercomputing systems.
- *Parallel System Software and Tools.* Tools, performance evaluation, development tools, run-time systems, libraries.
- *Parallel database systems.* Development of parallel database management systems for multiprocessor (multicore) computing systems. Methods and algorithms of parallel database processing. High performance data mining.
- *Supercomputing Education.* Practice and experience of supercomputing education. Innovative approaches on teaching of HPC and parallel computing technologies.

Contents

DOI: 10.14529/jsfi140301

Scientific Big Data Visualization: a Coupled Tools Approach

Antoni Artigues[1,2]***, Fernando M. Cucchietti***[1,3]***, Carlos Tripiana Montes***[1,4]***, David Vicente***[1,5]***, Hadrien Calmet***[1,6]***, Guillermo Marín***[1,7]***, Guillaume Houzeaux***[1,8]***, Mariano Vazquez***[1,9]

We designed and implemented a parallel visualisation system for the analysis of large scale time-dependent particle type data. The particular challenge we address is how to analyse a high performance computation style dataset when a visual representation of the full set is not possible or useful, and one is only interested in finding and inspecting smaller subsets that fulfil certain complex criteria. We used Paraview as the user interface, which is a familiar tool for many HPC users, runs in parallel, and can be conveniently extended. We distributed the data in a supercomputing environment using the Hadoop file system. On top of it, we run Hive or Impala, and implemented a connection between Paraview and them that allows us to launch programmable SQL queries in the database directly from within Paraview. The queries return a Paraview-native VTK object that fits directly into the Paraview pipeline. We find good scalability and response times. In the typical supercomputer environment (like the one we used for implementation) the queue and management system make it difficult to keep local data in between sessions, which imposes a bottleneck in the data loading stage. This makes our system most useful when permanently installed on a dedicated cluster.

Introduction

High performance computer simulations can now routinely reach the peta-scale, producing up to tens or even hundreds of GB of data per single time step. The visualisation of such large data poses many challenges: from the practical details (e.g. visualisation must be done on site as the data is too large to move), to high level cognitive questions (e.g. how much and what part of the data is enough to reach meaningful conclusions.)

By far, the most time consuming operation in a large scale visualisation application is I/O. In a survey of visualisation performance on six supercomputers, Childs et. al. [1] showed that I/O time can be up to two orders of magnitude larger than calculation and render time. Recently, Mitchell, Ahrens and Wang [2] proposed a hybrid approach in which data was distributed across an HPC cluster using the Hadoop Distributed File System (HDFS), and used Kit-Wares ParaView as the user interface. The geometrical locality of data allowed to distribute the information such that each ParaView server had local access (through Hadoop) to the data needed for the distributed rendering. The performance increased linearly with the number of nodes used, following the effective bandwidth available to ParaView. However, this approach would have difficulty handling data that has time dependent geometry, or that requires parallel distribution along more than one variable. The example that motivated us was particle-like data from

[1]Barcelona Supercomputing Center, Department of Computer Applications on Science and Engineering, C/Gran Capita 2–4, Edifici Nexus I, Barcelona 08034 (Spain)

[2]`antoni.artigues@bsc.es`

[3]`fernando.cucchietti@bsc.es`

[4]`carlos.tripiana@bsc.es`

[5]`david.vicente@bsc.es`

[6]`hadrien.calmet@bsc.es`

[7]`guillermo.marin@bsc.es`

[8]`guillaume.houzeaux@bsc.es`

[9]IIIA — CSIC, Bellaterra (Spain), `mariano.vazquez@bsc.es`

biomechanical simulations, which move around during the simulation time, and that have physical properties relevant for the visualisation but that do not correlate with the fixed geometrical information of the computational mesh.

We follow along the line of and propose a more flexible system, which allows to distribute the dataset simultaneously across many different variables, and that permits to interactively extract and visualise subsets that fulfil certain conditions. Specifically, we deploy the Apache Hive on top of Hadoop, and implement a set of plugins for ParaView users to interact directly through programmable actions. We have pre-programmed filters with typical questions like Where do the particles that ended up in a given region come from?, or find all particles and their trajectories that are of a given type and go through this region at a certain time. ParaView allows to configure the query regions or domains through the GUI, and the user does not need to enter any code. However, more complex questions can be configured only using the SQL-like language of Hive and integrated seamlessly in ParaView.

By using HDFS we are able to handle huge numbers of particles and time steps, and the coupling between ParaView and Hive allows us to interact, process, and visualise them in almost-real time. A simple geometrical distribution criteria would invariably produce unbalanced loads (e.g. at the initial time all particles are concentrated in the launch region). By not using HDFS directly but through Hive, our system allows us to distribute particles using criteria like physical properties or simply the number of particles. Another advantage of this setup is that the set of nodes running HDFS/Hive need not be the same used for Paraview. In fact, in principle they do not even need to be run on the same computer (although one can expect a latency cost in this configuration), similar to what can be seen in information visualisation software like Tableau [3].

In this paper we focus on outlining the architecture of the system as we implemented it for validation. We describe the separate tools we used, and what is required to couple them in a supercomputing environment. The dataset we used for testing is a simulation of the airflow through the human respiratory system performed by Alya [4] on the FERMI supercomputer, Italy. Although the set of filters we developed and tested were geared towards this biomechanical simulation, our system should be useful for all types of simulation that track particles moving in fluids or vector fields (e.g. supernova explosions or agent based modelling).

1. Background: The tools

1.1. Paraview

ParaView [5] is an open-source, multi-platform data analysis and visualization application. ParaView users can quickly build visualizations to analyze their data using qualitative and quantitative techniques. The data exploration can be done interactively in 3D or programmatically using ParaView's batch processing capabilities.

ParaView was developed to analyze extremely large datasets using distributed memory computing resources. It can be run on supercomputers to analyze datasets of petscale size as well as on laptops for smaller data. It has a client server architecture to facilitate remote visualization of datasets, and generates level of detail (LOD) models to maintain interactive frame rates for large datasets.

Paraview is developed in the C++ programming language, and it is possible to add new functionality to it through an extensive plugin mechanism that allows to:

- Add new readers, writers, filters

- Add custom GUI components such as toolbar buttons to perform common tasks
- Add new views in for display data

The plugins are based on the VTK library [6], an open-source system for 3D computer graphics, image processing and visualization. VTK consists of a C++ class library and several interpreted interface layers including Tcl/Tk, Java, and Python. With Paraview you can create a VTK filter library and integrate its functionallity into the Paraview data analisys pipeline with a custom GUI associated to this library.

1.2. Hadoop

Hadoop [7] is a complete open source software architecture designed to solve the problem of retrieving and analyzing large data sets with deep and computationally intensive operations, like clustering and targeting. Hadoop implements a distributed file system called HDFS (Hadoop Distributed File System) that is designed to be installed in a computer cluster. Each computer node on the cluster runs a Hadoop daemon (DataNode) that controls the local hard drives of the node as a part of the HDFS. Hadoop also runs a control daemon (NameNode) on one of the computer nodes, responsible for distributing and replicating files along the controlled hard drive nodes.

Hadoop allow users to execute MapReduce [8] jobs with the data stored in the HDFS. MapReduce is a functional-type programming model that allows to execute operations in parallel on the DataNodes, mapping the operations out to all of the Hadoop nodes and then reducing the results back into a single result set.

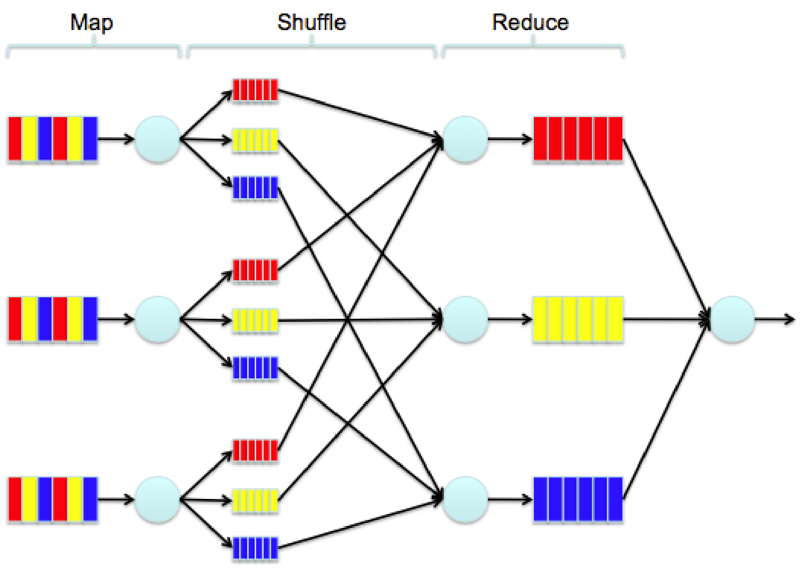


Figure 1. Example of MapReduce execution

1.3. Hive and other data warehouse software

Apache Hive [9] is an open source data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis. Hive is a data base engine that transforms SQL based queries [10] into MapReduce jobs for Hadoop clusters.

In general, complex database query problems are easier to solve writing them in SQL-like notation than creating MapReduce jobs to solve them. Furthermore, connecting applications with ODBC database drivers are easier than connecting to a Hadoop cluster, as Hive provides an abstraction layer that facilitates using Hadoop as a data warehouse system.

Hive offers tools to improve the queries performance taking advantage of the Hadoop architecture:

- Partitioning: Tables are linked to directories on HDFS with files in them. Without partitioning, Hive reads all the data in the directory and applies the query filters on it. On the other hand, with partitioning Hive stores the data in different directories and allows to read a reduced number of directories when it applies a query.
- Bucketing: By organizing tables or partitions into buckets, one can decrease the response time of join operations: A join of two tables that are bucketed on the same columns (which include the join columns) can be efficiently implemented as a map-side join.
- Map-joins: Hive allows to load the entire table in memory to speed up the joins (this only works for tables that are smaller than the computer main memory size).
- Parallel Execution: Hive is capable of parallelizing the execution of complex SQL queries by default.

Impala [11] is another data base engine that works on top of Hadoop and in combination with Hive. It has its own query solver that replaces the MapReduce jobs generated by Hive. The Impala query solver works faster than Hive but requires more RAM memory in the nodes. We have performed local tests (in a workstation) with Impala, showing a remarkable improvement in the response times of the system compared to using only Hive. However, we have not been yet able to adapt Impala to our supercomputer queue system. We plan to continue working to adapt Impala to our infrastructure in a near future.

1.4. The physical problem

The physical problem that led us to develop our tools is the simulation of air flow through the human respiratory system. Of particular medical interest is the simulation of Lagrangian particles (transported by the fluid) that can represent medicine or pollutants in the air.

1.4.1. The Alya system

Alya is the BSC in-house HPC-based multi-physics simulation code. It is designed from scratch to simulate highly complex problems seamlessly adapted to run efficiently onto high-end parallel supercomputers. The target domain is engineering, with all its particular features: complex geometries and unstructured meshes, coupled multi-physics with exotic coupling schemes and Physical models, ill-posed problems, flexibility needs for rapidly including new models, etc. Since its conception in 2004, Alya has shown scaling behaviour in an increasing number of cores. Alya is one of the twelve codes of the PRACE unified European benchmark suite (http://www.prace-ri.eu/ueabs), and therefore complies with the highest HPC European standards in terms of performance.

Two physical modules of Alya have been used to carry this work out. One the one hand, the incompressible Navier-Stokes equations are solved in a sniff condition, using a mesh of 350M elements. On the other hand, Lagrangian particles are transported by the fluid, by means of drag law and Brownian motion. All the details concerning the algorithms used in this work can be found in [12–15]. The simulations were performed on the Blue Gene Q supercomputer Fermi (CINECA, Italy), through a regular PRACE access call [16].

1.4.2. Numerical solution

The particular simulation output we worked with consists of a unique CSV file, with each row specifying the properties for one particle: time step, position, velocity, type, family, etc. A typical simulation generates a 300 Gb file. While the size can clearly be reduced by adopting a binary format, the actual limitation was the previous software (in house developed) used to analyze the data which could only handle a few thousand particles and time steps. For analysis purposes, having a larger data set directly correlates with better statistics and finer results. In fig. 2 we show a 3D representation of a typical simulation.

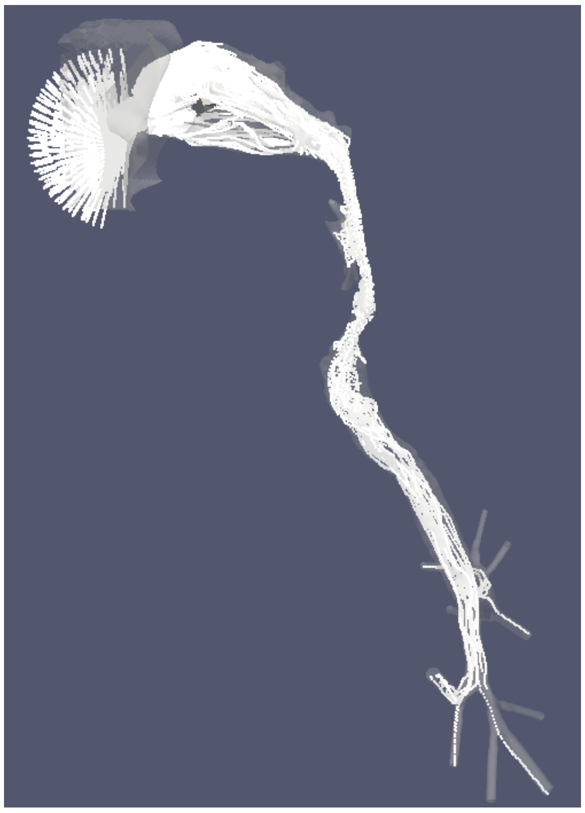


Figure 2. Example of the simulation results CSV file

Experts in fluid dynamics and nasal inspiratory flows were interviewed in order to make a list of the most important questions they would like to see answered using the simulation results:

- What is the path followed by a certain particle? We want to view in the 3D environment the path that followed a single (selectable) particle from the first to the last simulation time step.
- How are particles mixed in a certain region? We have different particles types: oxygen, co2, drugs, and we want to know how they are mixed in a certain region. We also need a visual representation and a concentration percentage information.
- Where do particles in certain region come from? Smell is a directional sense, and we have a delimited section inside our noses with sensors capable of smelling. The question aims to answer where do the particles that we smell come from.

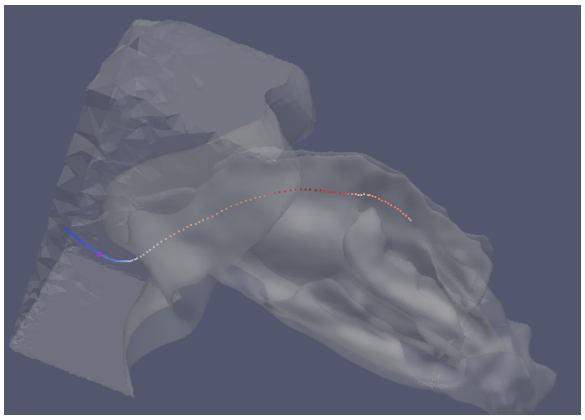


Figure 3. path followed by a certain particle

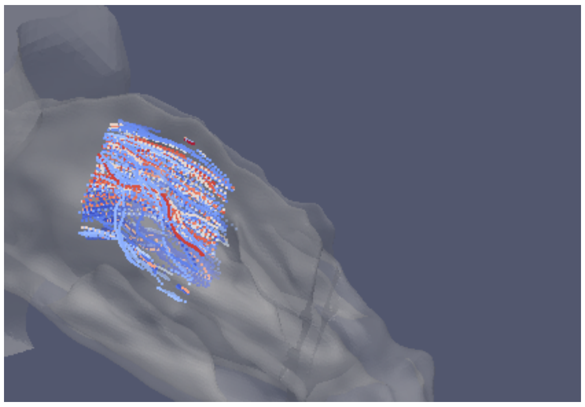


Figure 4. How are particles mixed in a certain region

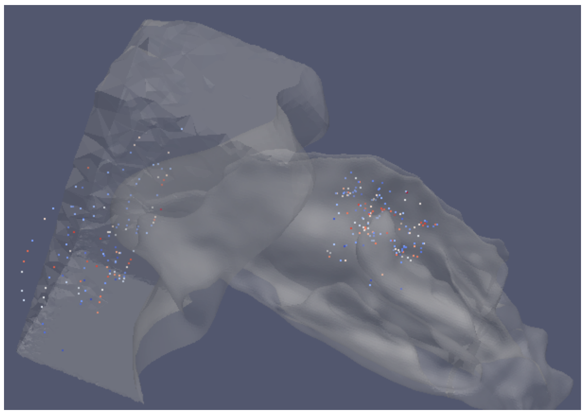


Figure 5. Where do particles in certain region come from?

- What are the paths followed by particles that end up in a given region? We want to analyze how the particles arrived to their final positions in the smell region of the nose. With this analysis we can detect certain strange loops that happen in the particle paths, and we can try to improve the performance of the medical devices that supply the drugs.
- How many particles go across a given section? We want to determine the throughput in arbitrary nose regions.
- Show only the particles that have a particular velocity or other physical properties in a certain direction.
- What are the particles deposited in certain nose parts? This analysis is for drug testing, because drugs have effects only when deposited in certain parts of the nose.

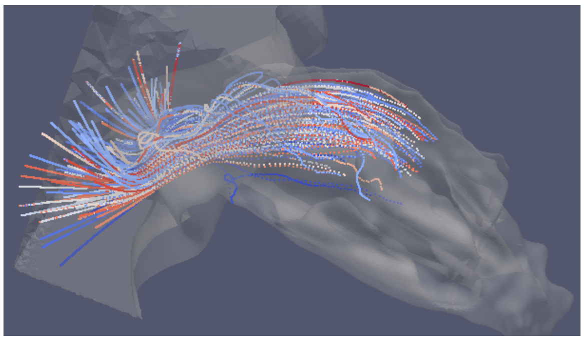


Figure 6. paths followed by the particles to a region

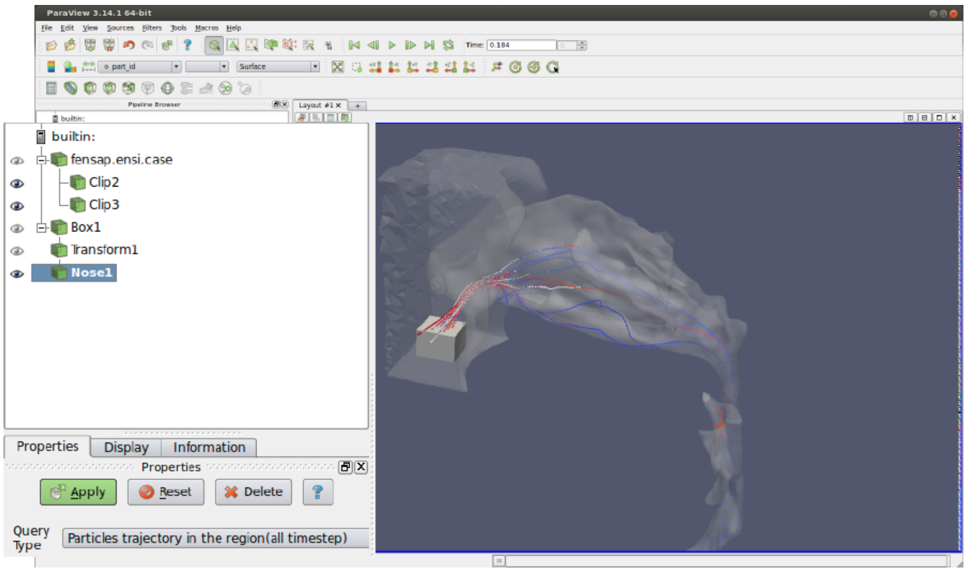
2. Requirements

The typical users of the application come from a biomedical background, are not experts in computational physics or high performance computing or IT. The application needs to be easy to use and avoid the complexity of the questions and its execution. Furthermore, it has to offer the results in a 3D, easy to understand, visual way.

The application has to offer tools to select precise 3D regions in a very accurate way.

To carry out long analysis, the system has to answer the questions in real or almost real time, so that the user can work interactively with the application.

When the data is too big to be rendered the application has to filter the results selecting only a relevant number of particles.



Query Type: Particles trajectory in the region(all timestep)

Figure 7. Paraview user interface to analyze results

The application has to provide an easy and systematic way to add more pre-programmed questions.

Our research center only provides supercomputer facilities [17] to execute tasks that need large computer power. The supercomputers have the advantage of having a very low latency time in the communication between the nodes at a very high transfer speed rate. Another advantage is that they have multiple connection networks between each node. However, the supercomputer also has two main requirements (which become constrains):

- Use of batch queue systems: Because all the executions, including the Hadoop daemons, have to be launched through a queue job, no permanent Hadoop and Hive installation can be deployed in the supercomputer.
- Only ssh connections are allowed: We can not use the ODBC protocol to connect the user's local computer with the Hadoop deployment on the supercomputer.

Due to this two restrictions, we must deploy the entire coupled tools *and* the full data set in the supercomputer each time that we need to analyze a simulation. Notice however that our system, as designed, can be easily installed in a permanent manner in a cluster without our supercomputer's restrictions. Therefore, we do not consider them a crucial disadvantage of the architecture (it is a disadvantage of our available resources for implementation).

From the user's point of view the steps to do the simulation analysis with our tools are:

- Load the geometric mesh, in our case the respiratory system, into Paraview.
- Add a box source to the pipeline.
- Scale and translate the box to the desired 3D region.
- Add a query plugin to the pipeline connected to the box.
- Select the desired query to execute from a list, for example: Obtain the particle's paths that are in the region and goes from outside to inside the nose.
- Click apply and obtain the results, in about 15 seconds depending on the query, on the 3D Paraview panel.

3. Architecture

3.1. Structure

In this section we describe how we coupled the different tools in order to solve the requirements described above.

As is shown in fig. 8, the main components of the structure are:

- Paraview: User interacts with Paraview to analyze the results
- NosePlugin: We have developed this Paraview plugin that allows users to query the simulation data.
- ODBC connector: Connects the NosePlugin with the data base.
- ETL NosePlugin: A script that loads all the simulations results into the database.
- Hive: The relational and distributed database engine in charge of executing SQL queries.
- Hadoop: The distributed file system used by Hive to execute the SQL questions in parallel through map-reduce jobs.
- Cluster: Hadoop works on top of a computer cluster, it needs a minimum of two nodes in order to run correctly.

3.2. Workflow

In this section we explain what are the steps required to solve a question about the simulation results.

From the users' point of view, they have to submit a job into the supercomputing queue system, specifying the path to the simulation results, and the number of computer nodes required. In our case the job is called infrastructureJob.

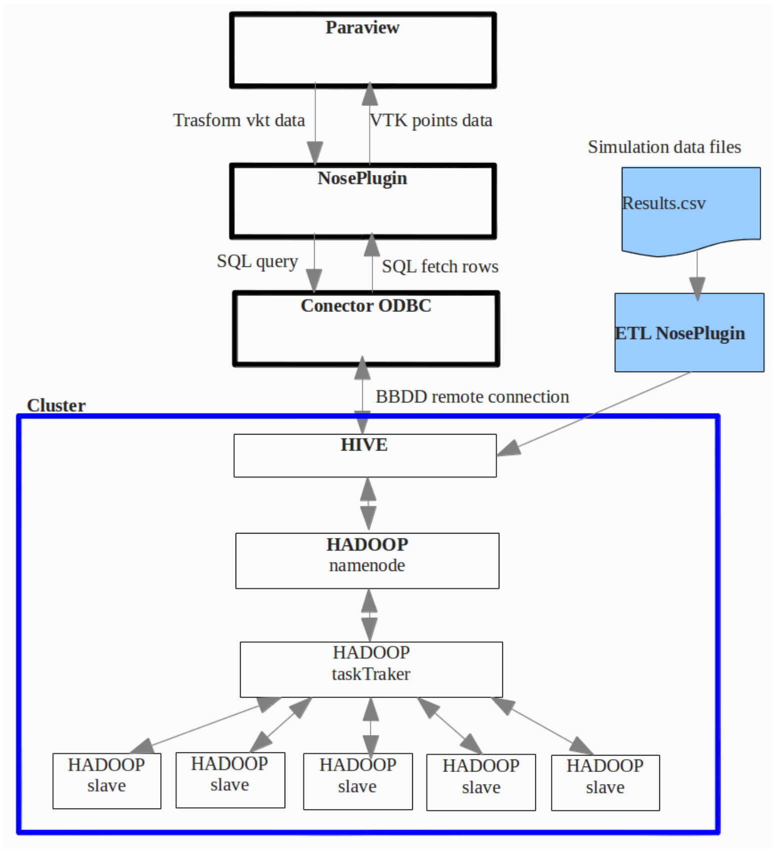


Figure 8. Coupled tools structure

Notice that the following first two steps are special and must be repeated every time the architecture is used only in a queue system that does not allow permament installations of software. In a dedicated cluster setup they are executed once at deployment and are not a crucial part of our system.

3.2.1. Deploy services

The first step is to deploy the Hadoop and Hive infrastructure in a subset of a supercomputer nodes, see fig. 9. The infrastructureJob:

- Allocates the required nodes and gets their IP addresses.
- Creates the required Hadoop folders in each of the nodes local hard drives.
- Creates the Hadoop configuration files to run Hadoop correctly in the allocated nodes.
- Configures the ODBC driver with the correct IP address
- Starts the Hadoop and Hive daemons on the nodes
- Starts the ETL NosePlugin Python script.

The deployment step requires a minimum of two nodes in order to work properly and three nodes to have data replication.

3.2.2. Load data

The ETL NosePlugin is a Python script that communicates with the data base thought a Hive ODBC connector and executes DDL and DML SQL sentences to load the simulation results into the data base.

The script executes these steps:

1. Prepare data: Checks the data file integrity and cleans unnecessary data.
2. Create database tables: Creates all the relational tables necessary to store the simulation results(see fig. 10)
3. Iterates over the CSV simulation results file
   (a) Gets next n lines from the file.
   (b) Generates a temporal file with these n lines.
   (c) Loads this temporal file data into a temporal relational table with the LOAD DATA command.
   (d) Loads the data into the final tables with a list of "INSERT SELECT" SQL statements.

In order to load big data files to the system we have done these optimizations:

- Adjust the partitions number to the data size
- Adapt Hadoop mapred-site.xml configuration to the supercomputer.
- Check SQL exceptions and try one more time when errors happen.
- Partition the main CSV file in slices
- Open and close connection with the database each time.

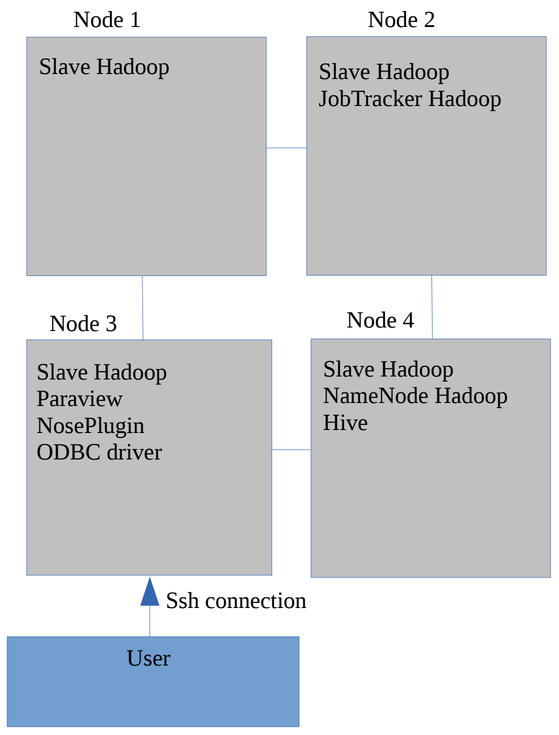


Figure 9. Coupled tools deployment in the Minotauro Supercomputer with 4 nodes

3.2.3. Start Paraview and NosePlugin

The last step of the noseInfrastructure job is to start Paraview with the NosePlugin loaded in it. The Paraview application is deployed in the same supercomputer nodes as Hadoop and Hive. We run the Paravier server in many nodes in parallel to have more rendering power when we render the query results.

3.2.4. Prepare Paraview to ask questions to the system

At this point the user can interact with the Paraview instance. The first step is to load the geometric nose mesh into Paraview. This mesh has to be small (meaning that it can be stored in the user's local file system). The next step is to add and position a box into the pipeline. Finally the user has to load the nosePlugin attached to the box.

3.3. Asking questions to the system

The user now can select a question from the nosePlugin GUI and click ”apply”. In that moment:

1. the nosePlugin receives the box position thought a VTK object data and the question to be asked to the data base in a string field.
2. The nosePlugin creates the SQL query and sends it to the data base through the ODBC Connector.
3. The Hadoop and Hive cluster is in charge of solving the query in real time.
4. The nosePlugin receives from the data base the SQL fetched rows corresponding to the answer.
5. The nosePlugin transforms the response into a VTK object and sends it to Paraview

4. Testing and validation

4.1. Validation

In order to do a basic test and validation we ran a small simulation that contains 2256 particles and 867 timesteps. This test case took only 184MB of disk storage. We installed the entire infrastructure in a standalone laptop and we ran the entire workflow on this machine.

We asked the system for the particles list in a well known region and time step, and we compared the obtained results with the expected response to validate the entire architecture.

4.2. Minotauro implementation

We deployed the architecture in the Minotauro supercomputer [18]. Minotauro has 126 compute nodes, each node has 12 cores, 24 GB of RAM memory, 12MB of cache memory and 250 GB flash local disk storage. Each also has two NVIDIA M2090 cards, each one with 512 CUDA Cores.

The HDFS was installed in the local flash disk of each node, the total available storage for Hadoop is 178.83 GB per node.

We developed a bash script in order to deploy the architecture in Minotauro each time that we needed to analyze results. The script is launched in the SLURM Minotauro queue system and installs Hadoop and Hive dinamically depending on the resources requested by the user and the nodes caught by the queue system.

4.3. Tests and scalability

Starting from the 184MB simulation response we have generated synthetic and controlled results of different file sizes: 15GB, 60GB, 120GB, 240GB and 575GB in order to test the scalability of the coupled tools.

4.3.1. Scalability test — number of nodes

The first scalability test consisted of:

1. Loading the 60GB file in 4, 8 and 16 nodes in Minotauro
2. Obtaining the particles that are inside a cube with the corresponding SQL query.
3. Measuring the time taken to resolve the query.

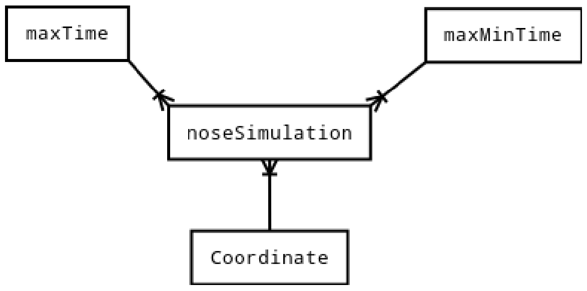


Figure 10. Tables database structure

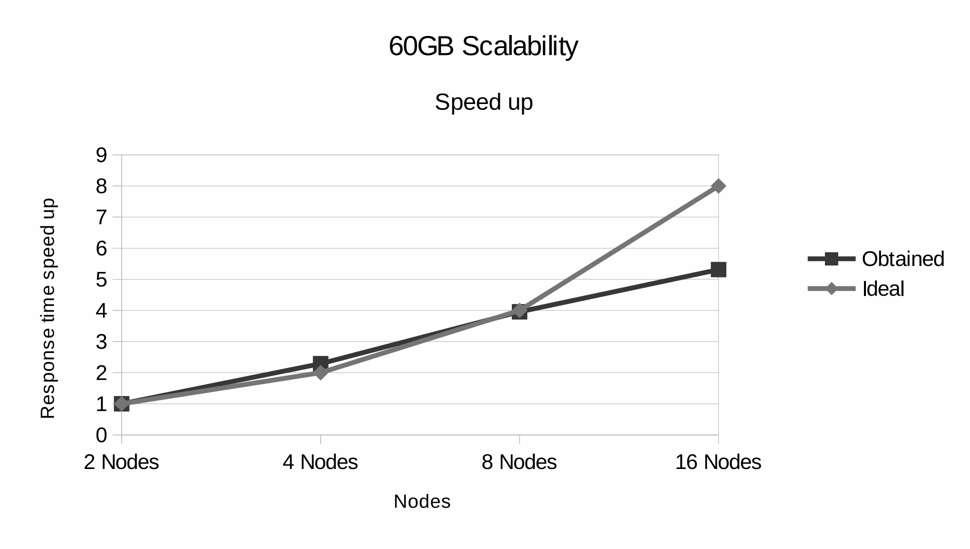


Figure 11. Scalability for the 60GB results file

Table 1. Scalability results for 60GB

Nodes	Response time	Speed up	Ideal
2	185.24 s	1	1
4	80.97 s	2.27	2
8	46.83 s	3.95	4
16	34.87 s	5.31	8

2.2878374314 3.9555077263 5.3116754452

The response times obtained in the 60GB scalability test are shown in fig. 11 and tab. 1. The scalability results show that incrementing the number of nodes increases the speed up in the response time: from 2 to 4 nodes the response time is better than the expected, however for 16 nodes the scalability is below the ideal, maybe caused by the communication overhead between the distributed Hadoop jobs.

4.3.2. Scalability test — size of the dataset

The second scalability tests consisted of:

1. Loading the 60GB and 120GB files in 8 Minotauro nodes.
2. Obtaining the particles that are inside a cube with the corresponding SQL query for both file sizes.
3. Measuring the time taken to resolve the query on each file.

The times we measured are:

Nodes	60GB file	120GB file
8	46.83 s	50.97 s

which suggests that our system scales well with the size of the dataset. The expected time for 120GB was 93.66 seconds, but we obtained 50.97 seconds. This confirms that the scalability is better than the expected before reaching the ideal number of nodes. For example, with the 60GB file from 2 two 8 nodes the performances are better than the expected, but for 16 nodes the response time is below ideal.

4.3.3. Optimization test

The optimization test is based on quantifying the speed-up obtained when we apply different Hive optimizations to the queries:

- Star data base schema
- Hive map joins
- Hive partitioning in time dimension
- Hive partitioning in space dimensions

We loaded a 15GB results file into 4 Minotauro nodes, we obtained the particles that are inside a cube with optimized and not-optimized queries respectively. The response time, in seconds, for both queries:

Nodes	not optimized	optimized
4	430.19 s	78.81 s

With Hive optimizations we obtained a speed up of 5.4x in our queries, emphasizing the fact that database optimization is a crucial aspect in all systems based on distributed technologies.

4.3.4. Impala test

In this test we compared the performance between Hive and Impala using a 184MB dataset. Because we could not install Impala in our supercomputers, we used a Cloudera Virtual Machine with Impala, Hive and Hadoop preinstalled on it. We obtained the particles that are inside a cube with the corresponding SQL query for both systems: Impala and Hive. The response times in seconds were:

Impala	Hive
0.712	15.043

4.4. User feedback

Our system has been tested and put to use by the biomedical researchers in charge of the physical simulation. From informal interviews and interactions with them, we have received positive feedback, in the sense that our tool has allowed them to improve their workflow considerably and will allow them to refine their observations (by increasing the number of particles they simulate). Users also point out the ease of designing new queries into the system, which they feel as a natural extension of Paraview capabilitites. They have identified two problems: the setting up stage is too long and inconvenient, and the response time, although short, is still away from real-time.

5. Conclusions

We have designed and implemented a system for the visualization and analysis of particle-type large sets of data using a coupled tools approach. Through testing we verified that the efficiency of the system scales well with the amount of resources used (e.g. number of compute nodes), meaning that it is a viable approach for larger (peta-scale) datasets where the tools we used have already been tested. Our test implementation would not scale up to peta-sized databases because of the constrains imposed by the supercomputer queue system (which would require days just to load the data set), but in a normal cluster environment with a permanent Hadoop and Hive installation this would not be a problem. Although the set of filters we developed and tested were geared towards our particular biomechanical simulation study case, our system should be useful for all types of simulation that track particles moving in fluids or vector fields (e.g. supernova explosions or agent based modelling). The problems detected by users will be resolved in the near future with a production version of our system installed permanently in a small cluster, where we will also be able to install and use Impala to improve the responsiveness of the queries. In the future we will investigate the parametrization of the trajectories in the data set into e.g. Bezier curves, thus moving the complexity of the data queries into computational complexity.

This work was partially supported by the Spanish Severo Ochoa Program, grant SEV-2011-00067. We acknowledge PRACE for awarding us access to the resource FERMI based in Italy at CINECA.

References

1. H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Maxi. A contract based system for large data visualization. In *Visualization, 2005. VIS 05. IEEE*, 2005.

2. Christopher Mitchell, James Ahrens, and Jun Wang. Visio: Enabling interactive visualization of ultra-scale, time series data via high-bandwidth distributed i/o systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011.

3. Tableau software. `http://www.tableausoftware.com/`. Tableau is developed by Tableau Software. `http://www.tableausoftware.com`.

4. Alya system - large scale computational mechanics. http://www.bsc.es/computer-applications/alya-system.

5. Paraview software. http://www.paraview.org. Paraview is developed by Kitware company. http://www.kitware.com.

6. VTK api. http://www.vtk.org. VTK is developed by Kitware company. http://www.kitware.com.

7. Hadoop. http://hadoop.apache.org/.

8. Yang Hung-Chih, Ali Dasdan, Hsiao Ruey-Lung, and D. Stott Parker. Map-reduce-merge simplified relational data processing on large clusters. In *Proc. of ACM SIGMOD*, 2007.

9. Hive database engine. http://hive.apache.org/.

10. Information technology database languages sql. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681. ISO/IEC 9075-1:2011 standard.

11. Impala database engine. http://impala.io. Impala is developed by Cloudera company. http://www.cloudera.com.

12. G. Houzeaux, R. Aubry, and M. Vázquez. Extension of fractional step techniques for incompressible flows: The preconditioned orthomin(1) for the pressure schur complement. *Computers & Fluids*, 44:297–313, 2011.

13. G. Houzeaux, R. de la Cruz, H. Owen, and M. Vázquez. Parallel uniform mesh multiplication applied to a navier-stokes solver. *Computers & Fluids*, 80:142–151, 7 2013.

14. G. Houzeaux, H. Owen, B. Eguzkitza, C. Samaniego, R. de la Cruz, H. Calmet, M. Vázquez, and M. Ávila. *Developments in Parallel, Distributed, Grid and Cloud Computing for Engineering*, volume 31 of *Computational Science, Engineering and Technology Series*, chapter Chapter 8: A Parallel Incompressible Navier-Stokes Solver: Implementation Issues, pages 171–201. Saxe-Coburg Publications, b.h.v. topping and p. iványi (Editor) edition, 2013.

15. M. Vázquez, G. Houzeaux, S. Koric, A. Artigues, J. Aguado-Sierra, R. Arís, D. Mira, H. Calmet, F. Cucchietti, H. Owen, A. Taha, and J.M. Cela. Alya: Towards exascale for engineering simulation codes. *arXiv.org*, 2014.

16. G. Houzeaux. Numerical simulation of air flow in the human large airways. PRACE 4th regular call. 20m core hours on tier-0, Fermi (I)., 2012.

17. Supercomputer description. http://en.wikipedia.org/wiki/Supercomputer.

18. Minotauro - nvidia gpu cluster. http://www.bsc.es/marenostrum-support-services/other-hpc-facilities/nvidia-gpu-cluster.

DOI: 10.14529/jsfi140302

Research Problems and Opportunities in Memory Systems

Onur Mutlu[1], *Lavanya Subramanian*[1]

The memory system is a fundamental performance and energy bottleneck in almost all computing systems. Recent system design, application, and technology trends that require more capacity, bandwidth, efficiency, and predictability out of the memory system make it an even more important system bottleneck. At the same time, DRAM technology is experiencing difficult *technology scaling* challenges that make the maintenance and enhancement of its capacity, energy-efficiency, and reliability significantly more costly with conventional techniques.

In this article, after describing the demands and challenges faced by the memory system, we examine some promising research and design directions to overcome challenges posed by memory scaling. Specifically, we describe three major *new* research challenges and solution directions: 1) enabling new DRAM architectures, functions, interfaces, and better integration of the DRAM and the rest of the system (an approach we call *system-DRAM co-design*), 2) designing a memory system that employs emerging non-volatile memory technologies and takes advantage of multiple different technologies (i.e., *hybrid memory systems*), 3) providing predictable performance and QoS to applications sharing the memory system (i.e., *QoS-aware memory systems*). We also briefly describe our ongoing related work in combating scaling challenges of NAND flash memory.

Keywords: memory systems, scaling, DRAM, flash, non-volatile memory, QoS, reliability.

Introduction

Main memory is a critical component of all computing systems, employed in server, embedded, desktop, mobile and sensor environments. Memory capacity, energy, cost, performance, and management algorithms must scale as we scale the size of the computing system in order to maintain performance growth and enable new applications. Unfortunately, such scaling has become difficult because recent trends in systems, applications, and technology greatly exacerbate the memory system bottleneck.

1. Memory System Trends

In particular, on the *systems/architecture front*, energy and power consumption have become key design limiters as the memory system continues to be responsible for a significant fraction of overall system energy/power [112]. More and increasingly heterogeneous processing cores and agents/clients are sharing the memory system [11, 36, 39, 60, 78, 79, 178, 181], leading to increasing demand for memory capacity and bandwidth along with a relatively new demand for predictable performance and quality of service (QoS) from the memory system [129, 137, 176].

On the *applications front*, important applications are usually very data intensive and are becoming increasingly so [17], requiring both real-time and offline manipulation of great amounts of data. For example, next-generation genome sequencing technologies produce massive amounts of sequence data that overwhelms memory storage and bandwidth requirements of today's high-end desktop and laptop systems [9, 111, 186, 196, 197] yet researchers have the goal of enabling low-cost personalized medicine, which requires even larger amounts of data and their effective analyses. Creation of new *killer applications* and usage models for computers likely depends on how well the memory system can support the efficient storage and manipulation of data in such

[1]Carnegie Mellon University, Pittsburgh, USA

data-intensive applications. In addition, there is an increasing trend towards consolidation of applications on a chip to improve efficiency, which leads to the sharing of the memory system across many heterogeneous applications with diverse performance requirements, exacerbating the aforementioned need for predictable performance guarantees from the memory system [176, 182].

On the *technology front*, two major trends profoundly affect memory systems. First, there is increasing difficulty scaling the well-established charge-based memory technologies, such as DRAM [4, 10, 70, 90, 97, 102, 124] and flash memory [20, 21, 24, 98, 123], to smaller technology nodes. Such scaling has enabled memory systems with reasonable capacity and efficiency; lack of it will make it difficult to achieve high capacity and efficiency at low cost. Challenges with DRAM scaling were recently highlighted by a paper written by Samsung and Intel [83]. Second, some emerging resistive memory technologies, such as phase change memory (PCM) [102, 103, 159, 163, 192], spin-transfer torque magnetic memory (STT-MRAM) [31, 100] or resistive RAM (RRAM) [193] appear more scalable, have latency and bandwidth characteristics much closer to DRAM than flash memory and hard disks, and are non-volatile with little idle power consumption. Such emerging technologies can enable new opportunities in system design, including, for example, the unification of memory and storage subsystems [127]. They have the potential to be employed as part of main memory, alongside or in place of less scalable and leaky DRAM, but they also have various shortcomings depending on the technology (e.g., some have cell endurance problems, some have very high write latency/power, some have low density) that need to be overcome or tolerated.

2. Memory System Requirements

System architects and users have always wanted more from the memory system: high performance (ideally, zero latency and infinite bandwidth), infinite capacity, all at zero cost! The aforementioned trends do not only exacerbate and morph the above requirements, but also add some new requirements. We classify the requirements from the memory system into two categories: *exacerbated traditional requirements* and *(relatively) new requirements*.

The traditional requirements of performance, capacity, and cost are greatly exacerbated today due to increased pressure on the memory system, consolidation of multiple applications/agents sharing the memory system, and difficulties in DRAM technology and density scaling. In terms of *performance*, two aspects have changed. First, today's systems and applications not only require low latency and high bandwidth (as traditional memory systems have been optimized for), but they also require new techniques to manage and control memory interference between different cores, agents, and applications that share the memory system [40, 129, 137, 176, 182] in order to provide high system performance as well as predictable performance (or quality of service) to different applications [176]. Second, there is a need for increased memory bandwidth for many applications as the placement of more cores and agents on chip make the memory pin bandwidth an increasingly precious resource that determines system performance [71], especially for memory-bandwidth-intensive workloads, such as GPGPUs [80, 81, 146], heterogeneous systems [11], and consolidated workloads [73, 74, 137]. In terms of *capacity*, the need for memory capacity is greatly increasing due to the placement of multiple data-intensive applications on the same chip and continued increase in the data sets of important applications. One recent work showed that given that the core count is increasing at a faster rate than DRAM capacity, the expected memory capacity per core is to drop by 30% every two years [113], an alarming trend since much of today's software innovations and features rely on increased memory capacity. In terms of *cost*, increasing difficulty in DRAM technology scaling poses a difficult challenge to building higher density (and, as a result, lower cost) main memory systems. Similarly, cost-effective options for providing high reliability and increasing memory bandwidth are needed to scale the systems proportionately with the reliability and data throughput needs of today's data-intensive applications. Hence, the three traditional requirements of performance, capacity, and cost have become exacerbated.

The relatively new requirements from the main memory system are threefold. First, *technology scalability*: there is a new need for finding a technology that is much more scalable than DRAM in terms of capacity, energy, and cost, as described earlier. As DRAM continued to scale well from the above-100-nm to 30-nm technology nodes, the need for finding a more scalable technology was not a prevalent problem. Today, with the significant circuit and device scaling challenges DRAM has been facing below the 30-nm node [83], it is. Second, there is a relatively new need for providing *performance predictability and QoS* in the shared main memory system. As single-core systems were dominant and memory bandwidth and capacity were much less of a shared resource in the past, the need for predictable performance was much less apparent or prevalent [129]. Today, with increasingly more cores/agents on chip sharing the memory system and increasing amounts of workload consolidation, memory fairness, predictable memory performance, and techniques to mitigate memory interference have become first-class design constraints. Third, there is a great need for much higher *energy/power/bandwidth efficiency* in the design of the main memory system. Higher efficiency in terms of energy, power, and bandwidth enables the design of much more scalable systems where main memory is shared between many agents, and can enable new applications in almost all domains where computers are used. Arguably, this is not a *new* need today, but we believe it is another first-class design constraint that has not been as traditional as performance, capacity, and cost.

3. Solution Directions and Research Opportunities

As a result of these systems, applications, and technology trends and the resulting requirements, it is our position that researchers and designers need to fundamentally rethink the way we design memory systems today to 1) overcome scaling challenges with DRAM, 2) enable the use of emerging memory technologies, 3) design memory systems that provide predictable performance and quality of service to applications and users. The rest of this article describes *our solution ideas* in these three relatively new research directions, with pointers to specific techniques when possible.[2] Since scaling challenges themselves arise due to difficulties in enhancing memory components at *solely* one level of the computing stack (e.g., the device and/or circuit levels in case of DRAM scaling), we believe effective solutions to the above challenges will require cooperation across different layers of the computing stack, from algorithms to software to microarchitecture to devices, as well as between different components of the system, including

[2]Note that this paper is *not* meant or designed to be a survey of *all* recent works in the field of memory systems. There are many such insightful works, but we do not have space in this paper to discuss them all. This paper *is* meant to outline the challenges and research directions in memory systems as *we* see them. Therefore, many of the solutions we discuss draw heavily upon *our own* past, current, and future research. We believe this will be useful for the community as the directions we have pursued and are pursuing are hopefully fundamental challenges for which other solutions and approaches would be greatly useful to develop. We look forward to similar papers from other researchers describing their perspectives and solution directions/ideas.

processors, memory controllers, memory chips, and the storage subsystem. As much as possible, we will give examples of such cross-layer solutions and directions.

4. New Research Challenge 1: New DRAM Architectures

DRAM has been the choice technology for implementing main memory due to its relatively low latency and low cost. DRAM process technology scaling has enabled lower cost per unit area by enabling reductions in DRAM cell size for a long time. Unfortunately, further scaling of DRAM cells has become costly [4, 10, 70, 83, 90, 102, 124] due to increased manufacturing complexity/cost, reduced cell reliability, and potentially increased cell leakage leading to high refresh rates. Recently, a paper by Samsung and Intel [83] has also discussed the key scaling challenges of DRAM at the circuit level. They have identified three major challenges as impediments to effective scaling of DRAM to smaller technology nodes: 1) the growing cost of refreshes [114], 2) increase in write latency, and 3) variation in the retention time of a cell over time [115]. In light of such challenges, we believe there are at least the following key issues to tackle in order to design new DRAM architectures that are much more scalable:

1) reducing the negative impact of refresh on energy, performance, QoS, and density scaling [28, 83, 86, 114, 115],

2) improving reliability of DRAM at low cost [86, 97, 122, 145],

3) improving DRAM parallelism/bandwidth [28, 96], latency [109, 110], and energy efficiency [96, 109, 114],

4) minimizing data movement between DRAM and processing elements, which causes high latency, energy, and bandwidth consumption, by doing more operations on the DRAM and the memory controllers [167],

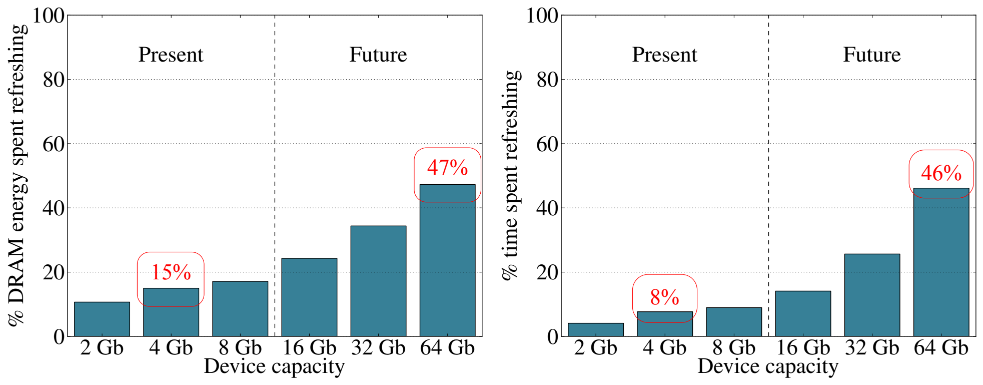
5) reducing the significant amount of waste in today's main memories in which much of the fetched/stored data can be unused due to coarse-granularity management [126, 153–155, 187, 199].

Traditionally, DRAM devices have been separated from the rest of the system with a rigid interface, and DRAM has been treated as a *passive* slave device that simply responds to the commands given to it by the memory controller. We believe the above key issues can be solved more easily if we rethink the DRAM architecture and functions, and redesign the interface such that DRAM, controllers, and processors closely cooperate. We call this high-level solution approach *system-DRAM co-design*. We believe key technology trends, e.g., the 3D stacking of memory and logic [5, 119, 188] and increasing cost of scaling DRAM solely via circuit-level approaches [70, 83, 90, 124], enable such a co-design to become increasingly more feasible. We proceed to provide several examples from our recent research that tackle the problems of refresh (and retention errors), parallelism, reliability, latency, energy efficiency, in-memory computation, and capacity and bandwidth waste.

4.1. Reducing Refresh Impact

With higher DRAM capacity, more cells need to be refreshed at likely higher rates than today. Our recent work [114] indicates that refresh rate limits DRAM density scaling: a hypothetical 64Gb DRAM device would spend 46% of its time and 47% of all DRAM energy for refreshing its rows, as opposed to typical 4Gb devices of today that spend 8% of the time and 15% of the DRAM energy on refresh (as shown in Figure 1). For instance, a modern supercomputer may have 1PB of memory in total [6]. If we assume this memory is built from 8Gb DRAM devices and a nominal refresh rate, 7.8kW of power would be expended, on average, just to refresh the entire 1PB memory. This is quite a large number, just to ensure the memory correctly keeps its contents! And, this power is *always* spent regardless of how much the supercomputer is utilized.

Today's DRAM devices refresh all rows at the same worst-case rate (e.g., every 64ms). However, only a small number of weak rows require a high refresh rate [86, 91, 115] (e.g., only ~1000 rows in 32GB DRAM require to be refreshed more frequently than every 256ms). Retention-Aware Intelligent DRAM Refresh (RAIDR) [114] exploits this observation: it groups DRAM rows into bins (implemented as Bloom filters [16] to minimize hardware overhead) based on the retention time of the weakest cell within each row. Each row is refreshed at a rate corresponding to its retention time bin. Since few rows need high refresh rate, one can use very few bins to achieve large reductions in refresh counts: our results show that RAIDR with three bins (1.25KB hardware cost) reduces refresh operations by ~75%, leading to significant improvements in system performance and energy efficiency as described by Liu et al. [114].



a) Power consumption b) Throughput Loss

Figure 1. Impact of refresh in current (DDR3) and projected DRAM devices. Reproduced from [114]

Like RAIDR, other approaches have also been proposed to take advantage of the retention time variation of cells across a DRAM chip. For example, some works proposed refreshing weak rows more frequently at a per-row granularity, others proposed not using memory rows with low retention times, and yet others suggested mapping critical data to cells with longer retention times so that critical data is not lost [7, 72, 89, 118, 149, 190] – see [114, 115] for a discussion of such techniques. Such approaches that exploit non-uniform retention times across DRAM require accurate retention time profiling mechanisms. Understanding of retention time as well as error behavior of DRAM devices is a critical research topic, which we believe can enable other mechanisms to tolerate refresh impact and errors at low cost. Liu et al. [115] provide an experimental characterization of retention times in modern DRAM devices to aid such understanding. Our initial results in that work, obtained via the characterization of 248 modern commodity DRAM chips from five different DRAM manufacturers, suggest that the retention time of cells in a modern device is largely affected by two phenomena: 1) Data Pattern Dependence, where the retention time of each DRAM cell is significantly affected by the data stored in other DRAM cells, 2) Variable Retention Time, where the retention time of a DRAM cell changes unpredictably over time. These two phenomena pose challenges against accurate and reliable determination of the retention time of DRAM cells, online or offline. A promising area of future research is to devise techniques that can identify retention times of DRAM cells in the presence of data pattern dependence and variable retention time. Khan et al.'s recent work [86] provides more analysis of the effectiveness of conventional error mitigation mechanisms for DRAM retention failures and proposes *online retention time profiling* as a solution for identifying retention times of DRAM cells as a potentially promising approach in future DRAM systems. We believe developing such system-level techniques that can detect and exploit DRAM characteristics online, during system operation, will be increasingly valuable as such characteristics will become much more difficult to accurately determine and exploit by the manufacturers due to the scaling of technology to smaller nodes.

4.2. Improving DRAM Reliability: Better DRAM Error Management

As DRAM technology scales to smaller node sizes, its reliability becomes more difficult to maintain at the circuit and device levels. In fact, we already have evidence of the difficulty of maintaining DRAM reliability from the DRAM chips operating in the field today. Our recent research [97] showed that a majority of the DRAM chips manufactured between 2010-2014 by three major DRAM vendors exhibit a particular failure mechanism called *row hammer*: by activating a row enough times within a refresh interval, one can corrupt data in nearby DRAM rows. The source code is available at [3]. This is an example of a *disturbance error* where the access of a cell causes disturbance of the value stored in a nearby cell due to cell-to-cell coupling (i.e., interference) effects, some of which are described by our recent works [97, 115]. Such interference-induced failure mechanisms are well-known in any memory that pushes the limits of technology, e.g., NAND flash memory (see Section 7). However, in case of DRAM, manufacturers have been quite successful in containing such effects until recently. Clearly, the fact that such failure mechanisms have become difficult to contain and that they have already slipped into the field shows that failure/error management in DRAM has become a significant problem. We believe this problem will become even more exacerbated as DRAM technology scales down to smaller node sizes. Hence, it is important to research both the (new) failure mechanisms in future DRAM designs as well as mechanisms to tolerate them. Towards this end, we believe it is critical to gather insights from the field by 1) experimentally characterizing DRAM chips using controlled testing infrastructures [86, 97, 110, 115], 2) analyzing large amounts of data from the field on DRAM failures in large-scale systems [164, 172, 173], 3) developing models for failures/errors based on these experimental characterizations, and 4) developing new mechanisms at the system and architecture levels that can take advantage of such models to tolerate DRAM errors.

Looking forward, we believe that increasing cooperation between the DRAM device and the DRAM controller as well as other parts of the system, including system software, would be greatly beneficial for identifying and tolerating DRAM errors. For example, such cooperation can enable the communication of information about *weak* (or, unreliable) cells and the characteristics of different rows or physical memory regions from the DRAM device to the system. The system can then use this information to optimize data allocation and movement, refresh rate management, and error tolerance mechanisms. Low-cost error tolerance mechanisms are likely to be enabled more efficiently with such coordination between DRAM and the system. In fact, as DRAM technology scales and error rates increase, it might become increasingly more difficult to maintain the illusion that DRAM is a perfect, error-free storage device (the *row hammer* failure mechanism [97] already provides evidence for this). DRAM may start looking increasingly like flash memory, where the memory controller manages errors so that an acceptable specified uncorrectable bit error rate is satisfied [20, 22]. We envision a *DRAM Translation Layer (DTL)*, not unlike the *Flash Translation Layer (FTL)* of today in spirit (which is decoupled from the processor and performs a wide variety of management functions for flash memory, including error correction, garbage collection, read/write scheduling, data mapping, etc.), can enable better scaling of DRAM memory into the future by not only enabling easier error management but also opening up new opportunities to perform computation, mapping and metadata management close to memory. This can become especially feasible in the presence of the trend of combining the DRAM controller and DRAM via 3D stacking. What should the interface be to such a layer and what should be performed in the DTL are promising areas of future research.

4.3. Improving DRAM Parallelism

A key limiter of DRAM parallelism is bank conflicts. Today, a bank is the finest-granularity independently accessible memory unit in DRAM. If two accesses go to the same bank, one has to *completely* wait for the other to finish before it can be started (see Figure 2). We have recently developed mechanisms, called SALP (subarray level parallelism) [96], that exploit the internal subarray structure of the DRAM bank (Figure 2) to *mostly* parallelize two requests that access the same DRAM bank. The key idea is to reduce the hardware sharing between DRAM subarrays so that accesses to the same bank but different subarrays can be initiated in a pipelined manner. This mechanism requires the exposure of the internal subarray structure of a DRAM bank to the controller and the design of the controller to take advantage of this structure. Our results show significant improvements in performance and energy efficiency of main memory due to parallelization of requests and improvement of row buffer hit rates (as row buffers of different subarrays can be kept active) at a low DRAM area overhead of 0.15%. Exploiting SALP achieves most of the benefits of increasing the number of banks at much lower area and power overhead. Exposing the subarray structure of DRAM to other parts of the system, e.g., to system software or memory allocators, can enable data placement and partitioning mechanisms that can improve performance and efficiency even further.

Note that other approaches to improving DRAM parallelism especially in the presence of refresh and long write latencies are also promising to be investigated. Chang et al. [28] discuss mechanisms to improve the parallelism between refreshes and reads/write requests, and Kang et al. [83] discuss the use of SALP as a promising method to tolerate long write latencies to DRAM, which they identify as one of the three key scaling challenges for DRAM, in addition to refresh and variable retention time. We refer the reader to these works for more information about the proposed parallelization techniques.

4.4. Reducing DRAM Latency and Energy

The DRAM industry has so far been primarily driven by the cost-per-bit metric: provide maximum capacity for a given cost. As shown in Figure 3, DRAM chip capacity has increased by approximately 16x in 12 years while the DRAM latency reduced by only approximately 20%. This is the result of a deliberate choice to maximize capacity of a DRAM chip while minimizing its cost. We believe this choice needs to be revisited in the presence of at least two key trends. First, DRAM latency is becoming more important especially for response-time critical workloads that require QoS guarantees [45]. Second, DRAM capacity is becoming very hard to scale and as

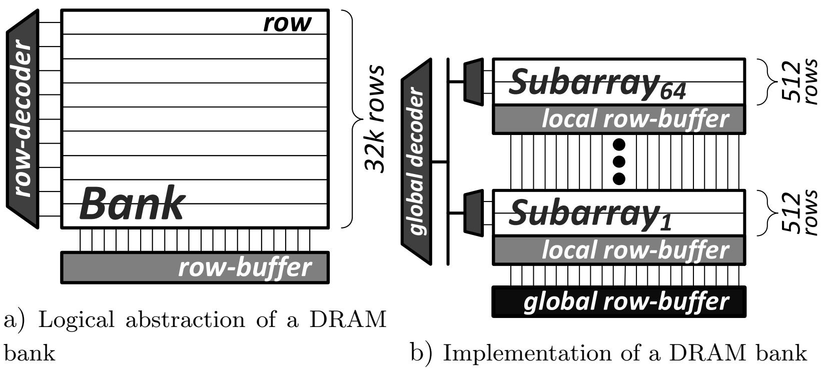


Figure 2. DRAM Bank Organization. Reproduced from [96]

a result manufacturers likely need to provide new values for the DRAM chips, leading to more incentives for the production of DRAMs that are optimized for objectives other than mainly capacity maximization.

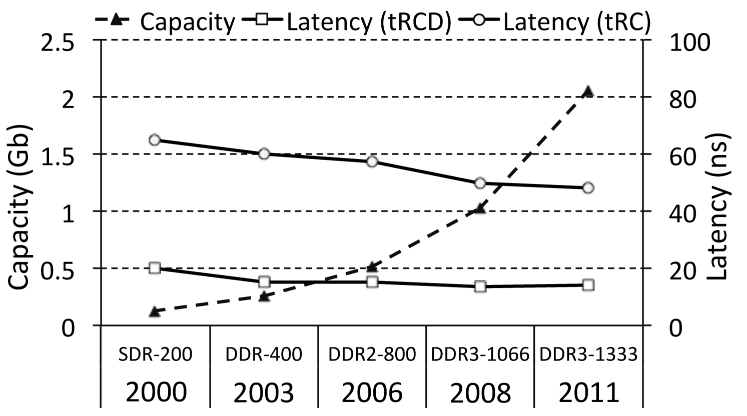


Figure 3. DRAM Capacity & Latency Over Time. Reproduced from [109]

To mitigate the high area overhead of DRAM sensing structures, commodity DRAMs (shown in Figure 4a) connect many DRAM cells to each sense-amplifier through a wire called a bitline. These bitlines have a high parasitic capacitance due to their long length, and this bitline capacitance is the dominant source of DRAM latency. Specialized low-latency DRAMs (shown in Figure 4b) use shorter bitlines with fewer cells, but have a higher cost-per-bit due to greater sense-amplifier area overhead. We have recently shown that we can architect a heterogeneous-latency bitline DRAM, called Tiered-Latency DRAM (TL-DRAM) [109], shown in Figure 4c, by dividing a long bitline into two shorter segments using an isolation transistor: a low-latency segment can be accessed with the latency and efficiency of a short-bitline DRAM (by turning off the isolation transistor that separates the two segments) while the high-latency segment enables high density, thereby reducing cost-per-bit. The additional area overhead of TL-DRAM is approximately 3% over commodity DRAM. Significant performance and energy improvements can be achieved by exposing the two segments to the memory controller and system software such that appropriate data is cached or allocated into the low-latency segment. We expect such approaches that design and exploit heterogeneity to enable/achieve the best of multiple worlds [132] in the memory system can lead to other novel mechanisms that can overcome difficult contradictory tradeoffs in design.

A recent paper by Lee et al. [110] exploits the extra margin built into DRAM timing parameters to reliably reduce DRAM latency when such extra margin is really not necessary (e.g.,

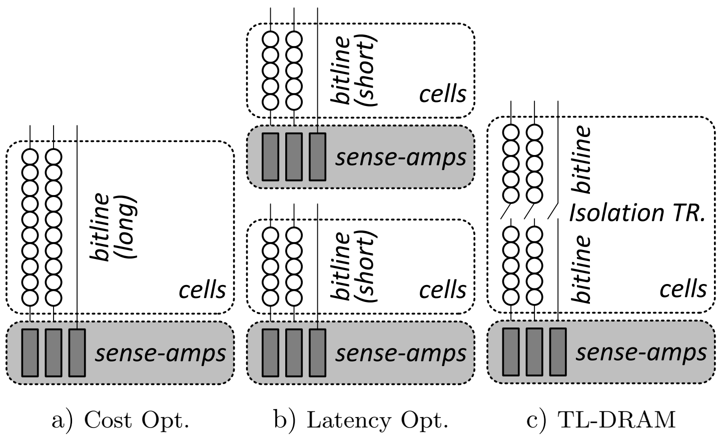


Figure 4. Cost Optimized Commodity DRAM (a), Latency Optimized DRAM (b), Tiered-Latency DRAM (c). Reproduced from [109]

when the operating temperature is low). The standard DRAM timing constraints are designed to ensure correct operation for the cell with the *lowest retention time* at the *highest acceptable operating temperature*. Lee et al. [110] make the observation that a significant majority of DRAM modules do not exhibit the worst case behavior and that most systems operate at a temperature much lower than the highest acceptable operating temperature, enabling the opportunity to significantly reduce the timing constraints. They introduce Adaptive-Latency DRAM (AL-DRAM), which dynamically measures the operating temperature of each DIMM and employs timing constraints optimized for that DIMM at that temperature. Results of their profiling experiments on 82 modern DRAM modules show that AL-DRAM can reduce the DRAM timing constraints by an average of 43% and up to 58%. This reduction in latency translates to a 14% average improvement in overall system performance across a wide variety of applications on the evaluated real systems. We believe such approaches to reducing memory latency (and energy) by exploiting *common-case device characteristics and operating conditions* are very promising: instead of *always* incurring the worst-case latency and energy overheads due to homogeneous, one-size-fits-all parameters, adapt the parameters dynamically to fit the common-case operating conditions.

Another promising approach to reduce DRAM energy is the use of dynamic voltage and frequency scaling (DVFS) in main memory [44, 46]. David et al. [44] make the observation that at low memory bandwidth utilization, lowering memory frequency/voltage does not significantly alter memory access latency. Relatively recent works have shown that adjusting memory voltage and frequency based on predicted memory bandwidth utilization can provide significant energy savings on both real [44] and simulated [46] systems. Going forward, memory DVFS can enable dynamic heterogeneity in DRAM channels, leading to new customization and optimization mechanisms. Also promising is the investigation of more fine-grained power management methods within the DRAM rank and chips for both active and idle low power modes.

4.5. Exporting Bulk Data Operations to DRAM: Enabling In-Memory Computation

Today's systems waste significant amount of energy, DRAM bandwidth and time (as well as valuable on-chip cache space) by sometimes unnecessarily moving data from main memory to processor caches. One example of such wastage sometimes occurs for bulk data copy and initialization operations in which a page is copied to another or initialized to a value. If the copied or initialized data is not immediately needed by the processor, performing such operations within DRAM (with relatively small changes to DRAM) can save significant amounts of energy, bandwidth, and time. We observe that a DRAM chip internally operates on bulk data at a row granularity. Exploiting this internal structure of DRAM can enable page copy and initialization to be performed entirely within DRAM without bringing any data off the DRAM chip, as we have shown in recent work [167]. If the source and destination page reside within the same DRAM subarray, our results show that a page copy can be accelerated by more than an order of magnitude (~11 times), leading to an energy reduction of ~74 times and *no* wastage of DRAM data bus bandwidth [167]. The key idea is to capture the contents of the source row in the sense amplifiers by 1) activating the row, then 2) *deactivating* the source row (using a new command which introduces very little hardware cost, amounting to less than 0.03% of DRAM chip area), and 3) immediately activating the destination row, which causes the sense amplifiers to drive their contents into the destination row, effectively accomplishing the page copy (shown at a high level in fig. 5). Doing so reduces the latency of a 4KB page copy operation from ~1000ns to less than 100ns in an existing DRAM chip. Applications that have significant page copy and initialization experience large system performance and energy efficiency improvements [167]. Future software can be designed in ways that can take advantage of such fast page copy and initialization operations, leading to benefits that may not be apparent in today's software that tends to minimize such operations due to their current high cost.

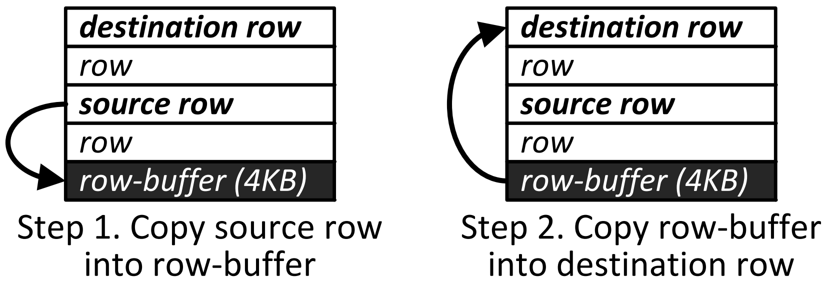


Figure 5. High-level idea behind RowClone's in-DRAM page copy mechanism

Going forward, we believe acceleration of other bulk data movement and computation operations in or very close to DRAM, via similar low-cost architectural support mechanisms, can enable promising savings in system energy, latency, and bandwidth. Given the trends and requirements described in Section 1, it is time to re-examine the partitioning of computation between processors and DRAM, treating memory as a first-class accelerator as an integral part of a heterogeneous parallel computing system [132].

4.6. Minimizing Memory Capacity and Bandwidth Waste

Storing and transferring data at large granularities (e.g., pages, cache blocks) within the memory system leads to large inefficiency when most of the large granularity is not needed [82, 101, 125, 126, 157, 166, 187, 199, 200]. In addition, much of the data stored in memory has significant redundancy [8, 13, 52, 59, 153–155, 198]. Two promising research directions are to develop techniques that can 1) efficiently provide fine granularity access/storage when enough and large granularity access/storage only when needed, 2) efficiently compress data in main memory and caches without significantly increasing latency and system complexity. Our results with new low-cost, low-latency cache compression [153] and memory compression [154] techniques and frameworks are promising, providing high compression ratios at low complexity and latency. For example, the key idea of *Base-Delta-Immediate compression* [153] is that many cache blocks have low dynamic range in the values they store, i.e., the differences between values stored in the cache block are small. Such a cache block can be encoded using a base value and an array of much smaller (in size) differences from that base value, which together occupy much less space than storing the full values in the original cache block. This compression algorithm has low decompression latency as the cache block can be reconstructed using a vector addition (or potentially even vector concatenation). It reduces memory bandwidth requirements, better utilizes memory/cache space, while minimally impacting the latency to access data. Granularity management and data compression support can potentially be integrated into DRAM controllers or partially provided within DRAM, and such mechanisms can be exposed to software, which can enable higher energy savings and higher performance improvements. Management techniques for compressed caches and memories (e.g., [155]) as well as flexible granularity memory system designs, software techniques/designs to take better advantage of cache/memory compression and flexible-granularity, and techniques to perform computations on compressed memory data are quite promising directions for future research.

4.7. Co-Designing DRAM Controllers and Processor-Side Resources

Since memory bandwidth is a precious resource, coordinating the decisions made by processor-side resources better with the decisions made by memory controllers to maximize memory bandwidth utilization and memory locality is a promising area of more efficiently utilizing DRAM. Lee et al. [106] and Stuecheli et al. [175] both show that orchestrating last-level cache writebacks such that dirty cache lines to the same row are evicted together from the cache improves DRAM row buffer locality of write accesses, thereby improving system performance. Going forward, we believe such coordinated techniques between the processor-side resources and memory controllers will become increasingly more effective as DRAM bandwidth becomes even more precious. Mechanisms that predict and convey slack in memory requests [42, 43], that orchestrate the on-chip scheduling of memory requests to improve memory bank parallelism [108] and that reorganize cache metadata for more efficient bulk (DRAM row granularity) tag lookups [168] can also enable more efficient memory bandwidth utilization.

5. New Research Challenge 2: Emerging Memory Technologies

While DRAM technology scaling is in jeopardy, some emerging technologies seem more scalable. These include phase-change memory PCM, spin-transfer torque magnetoresistive RAM (STT-MRAM) and resistive RAM (RRAM). These emerging technologies usually provide a tradeoff, and seem unlikely to *completely* replace DRAM (evaluated in [102–104] for PCM and in [100] for STT-MRAM), as they are not strictly superior to DRAM. For example, PCM is advantageous over DRAM because it 1) has been demonstrated to scale to much smaller feature sizes [102, 163, 192] and can store multiple bits per cell [202, 203], promising higher density, 2) is non-volatile and as such requires no refresh (which is a key scaling challenge of DRAM as we discussed in Section 4.1), and 3) has low idle power consumption. On the other hand, PCM has significant shortcomings compared to DRAM, which include 1) higher read latency and read energy, 2) *much* higher write latency and write energy, 3) limited endurance for a given PCM cell, a problem that does not exist (practically) for a DRAM cell, and 4) potentially difficult-to-handle reliability issues, such as the problem of *resistance drift* [165]. As a result, an important research challenge is how to utilize such emerging technologies at the system and architecture levels so that they can augment or perhaps even replace DRAM.

Our initial experiments and analyses [102–104] that evaluated the complete replacement of DRAM with PCM showed that one would require reorganization of peripheral circuitry of PCM chips (with the goal of absorbing writes and reads before they update or access the PCM cell array) to enable PCM to get close to DRAM performance and efficiency. These initial results are reported in Lee et al. [102–104] and they show that the performance, energy, and endurance of PCM chips can be greatly improved with the proposed techniques. We have also reached a similar conclusion upon evaluation of the complete replacement of DRAM with STT-MRAM [100]: reorganization of peripheral circuitry of STT-MRAM chips (with the goal of minimizing the number of writes to the STT-MRAM cell array, as write operations are high-latency and high-energy in STT-MRAM) enables an STT-MRAM based main memory to be more energy-efficient than a DRAM-based main memory.

One can achieve more efficient designs of PCM (or STT-MRAM) chips by taking advantage of the non-destructive nature of reads, which enables simpler and narrower row buffer organizations [125]. Unlike in DRAM, the entire memory row does not need to be buffered in a device where reading a memory row does not destroy the data stored in the row. Meza et al. [125] show that having narrow row buffers in emerging non-volatile devices can greatly reduce main memory dynamic energy compared to a DRAM baseline with large row sizes, without greatly affecting endurance, and for some NVM technologies, lead to improved performance. Going forward, designing systems, memory controllers and memory chips taking advantage of the specific property of non-volatility of emerging technologies seems promising.

We believe emerging technologies enable at least three major system-level opportunities that can improve overall system efficiency: 1) hybrid main memory systems, 2) non-volatile main memory, 3) merging of memory and storage. We briefly touch upon each.

5.1. Hybrid Main Memory

A hybrid main memory system [29, 47, 126, 156, 159, 162, 201] consists of multiple different technologies or multiple different types of the same technology with differing characteristics, e.g., performance, cost, energy, reliability, endurance. A key question is how to manage data allocation and movement between the different technologies so that one can achieve the best of (or close to the best of) the desired performance metrics. In other words, we would like to exercise the advantages of each technology as much as possible while hiding the disadvantages of any technology. Potential technologies include DRAM, 3D-stacked DRAM, embedded DRAM, PCM, STT-MRAM, other resistive memories, flash memory, forms of DRAM that are optimized for different metrics and purposes, etc. An example hybrid main memory system consisting of a large amount of PCM as main memory and a small amount of DRAM as its cache is depicted in Figure 6.

The design space of hybrid memory systems is large, and many potential questions exist. For example, should all memories be part of main memory or should some of them be used as a cache of main memory (or should there be configurability)? What technologies should be software visible? What component of the system should manage data allocation and movement? Should these tasks be done in hardware, software, or collaboratively? At what granularity should

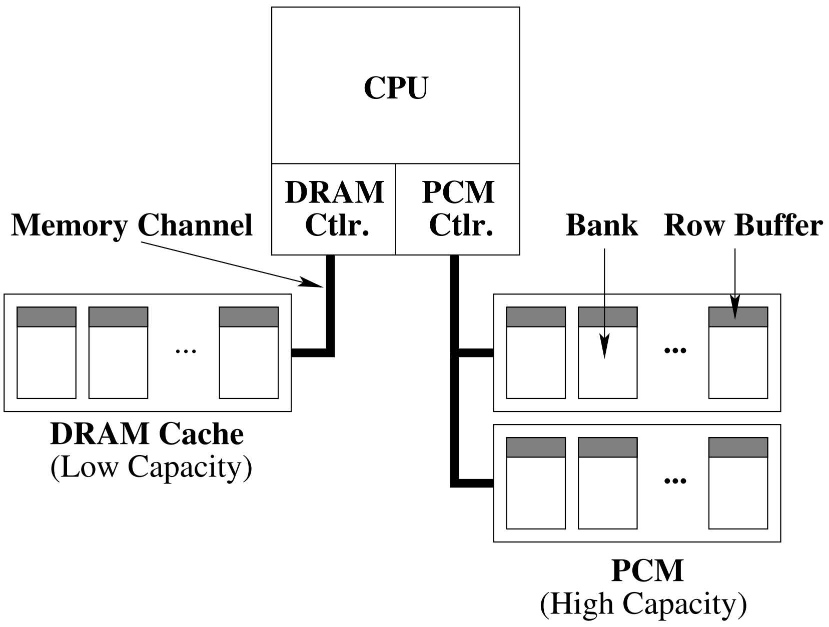


Figure 6. An example hybrid main memory system organization using PCM and DRAM chips. Reproduced from [201]

data be moved between different memory technologies? Some of these questions are tackled in [29, 47, 126, 156, 159, 162, 201], among other works recently published in the computer architecture community. For example, Yoon et al. [201] make the key observation that row buffers are present in both DRAM and PCM (see fig. 6), and they have (or can be designed to have) the same latency and bandwidth in both DRAM and PCM. Yet, row buffer misses are much more costly in terms of latency, bandwidth, and energy in PCM than in DRAM. To exploit this, they devise a policy that avoids accessing in PCM data that frequently causes row buffer misses. Hardware or software can dynamically keep track of such data and allocate/cache it in DRAM while keeping data that frequently hits in row buffers in PCM. PCM also has much higher write latency/power than read latency/power: to take this into account, the allocation/caching policy is biased such that pages that are written to more likely stay in DRAM [201].

Note that hybrid memory does not need to consist of completely different underlying technologies. A promising approach is to combine multiple different DRAM chips, optimized for different purposes. For example, recent works proposed the use of low-latency and high-latency DIMMs in separate memory channels and allocating performance-critical data to low-latency DIMMs to improve performance and energy-efficiency at the same time [29], or the use of highly-reliable DIMMs (protected with ECC) and unreliable DIMMs in separate memory channels and allocating error-vulnerable data to highly-reliable DIMMs to maximize server availability while minimizing server memory cost [122]. We believe these approaches are quite promising for scaling the DRAM technology into the future by specializing DRAM chips for different purposes. These approaches that exploit heterogeneity *do* increase system complexity but that complexity may be warranted if it is lower than the complexity of scaling DRAM chips using the same optimization techniques the DRAM industry has been using so far.

5.2. Making Non-volatile Main Memory Reliable and Secure

Non-volatility of main memory opens up new opportunities that can be exploited by higher levels of the system stack to improve performance and reliability/consistency (see, for example, [38, 48]). Researching how to adapt applications and system software to utilize fast, byte-addressable non-volatile main memory is an important research direction to pursue [127].

On the flip side, the same non-volatility can lead to potentially unforeseen security and privacy issues: critical and private data can persist long after the system is powered down [33], and an attacker can take advantage of this fact. Wearout issues of emerging technology can also cause attacks that can intentionally degrade memory capacity in the system [158, 171]. Securing non-volatile main memory is therefore an important systems challenge.

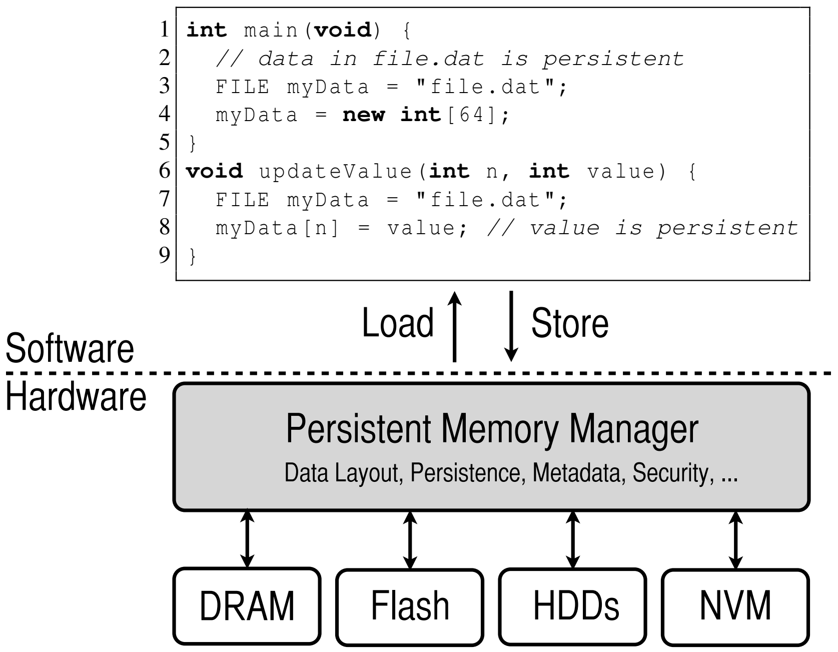
5.3. Merging of Memory and Storage

One promising opportunity fast, byte-addressable, non-volatile emerging memory technologies open up is the design of a system and applications that can manipulate *persistent data directly in memory* instead of going through a slow storage interface. This can enable not only much more efficient systems but also new and more robust applications. We discuss this opportunity in more detail below.

Traditional computer systems have a two-level storage model: they access and manipulate 1) volatile data in main memory (DRAM, today) with a fast load/store interface, 2) persistent data in storage media (flash and hard disks, today) with a slower file system interface. Unfortunately, such a decoupled memory/storage model managed via vastly different techniques (fast, hardware-accelerated memory management units on one hand, and slow operating/file system (OS/FS) software on the other) suffers from large inefficiencies in locating data, moving data, and translating data between the different formats of these two levels of storage that are accessed via two vastly different interfaces, leading to potentially large amounts of wasted work and energy [127, 170]. The two different interfaces arose largely due to the large discrepancy in the access latencies of conventional technologies used to construct volatile memory (DRAM) and persistent storage (hard disks and flash memory).

Today, new non-volatile memory technologies (NVM), e.g, PCM, STT-MRAM, RRAM, show the promise of storage capacity and endurance similar to or better than flash memory at latencies comparable to DRAM. This makes them prime candidates for providing applications a persistent *single-level store* with a single load/store-like interface to access all system data (including volatile and persistent data). In fact, if we keep the traditional two-level memory/storage model in the presence of these fast NVM devices as part of storage, the operating system and file system code for locating, moving, and translating persistent data from the non-volatile NVM devices to volatile DRAM for manipulation purposes becomes a great bottleneck, causing most of the memory energy consumption and degrading performance by an order of magnitude in some data-intensive workloads, as we showed in recent work [127]. With energy as a key constraint, and in light of modern high-density NVM devices, a promising research direction is to unify and coordinate the management of volatile memory and persistent storage in a single level, to eliminate wasted energy and performance, and to simplify the programming model at the same time.

To this end, Meza et al. [127] describe the vision and research challenges of a persistent memory manager (PMM), a hardware acceleration unit that coordinates and unifies memory/storage management in a single address space that spans potentially multiple different memory technologies (DRAM, NVM, flash) via hardware/software cooperation. Figure 7 depicts an example PMM, programmed using a load/store interface (with persistent objects) and managing an array of heterogeneous devices.



```
int main(void) {
  // data in file.dat is persistent
  FILE myData = "file.dat";
  myData = new int[64];
}
void updateValue(int n, int value) {
  FILE myData = "file.dat";
  myData[n] = value; // value is persistent
}
```

Load
Store
Software
Hardware
Persistent Memory Manager
Data Layout, Persistence, Metadata, Security, ...
DRAM
Flash
HDDs
NVM

Figure 7. An example Persistent Memory Manager (PMM). Reproduced from [127]

The spirit of the PMM unit is much like the virtual memory management unit of a modern virtual memory system used for managing working memory, but it is fundamentally different in that it redesigns/rethinks the virtual memory and storage abstractions and unifies them in a different interface supported by scalable hardware mechanisms. The PMM: 1) exposes a load/store interface to access persistent data, 2) manages data placement, location, persistence semantics, and protection (across multiple memory devices) using both dynamic access information and hints from the application and system software, 3) manages metadata storage and retrieval, needed to support efficient location and movement of persistent data, and 4) exposes hooks and interfaces for applications and system software to enable intelligent data placement and persistence management. Our preliminary evaluations show that the use of such a unit, if scalable and efficient, can greatly reduce the energy inefficiency and performance overheads of the two-level storage model, improving both performance and energy-efficiency of the overall system, especially for data-intensive workloads [127].

We believe there are challenges to be overcome in the design, use, and adoption of such a unit that unifies working memory and persistent storage. These challenges include:

1) How to devise efficient and scalable data mapping, placement, and location mechanisms (which need to be hardware/software cooperative).
2) How to ensure that the consistency and protection requirements of different types of data are adequately, correctly, and reliably satisfied (One example recent work tackled the problem of providing storage consistency at high performance [121]). How to enable the reliable and effective coexistence and manipulation of volatile and persistent data.

3) How to redesign applications such that they can take advantage of the unified memory/storage interface and make the best use of it by providing appropriate hints for data allocation and placement to the persistent memory manager.
4) How to provide efficient and high-performance backward compatibility mechanisms for enabling and enhancing existing memory and storage interfaces in a single-level store. These techniques can seamlessly enable applications targeting traditional two-level storage systems to take advantage of the performance and energy-efficiency benefits of systems employing single-level stores. We believe such techniques are needed to ease the software transition to a radically different storage interface.
5) How to design system resources such that they can concurrently handle applications/access-patterns that manipulate persistent data as well as those that manipulate non-persistent data. (One example recent work [204] tackled the problem of designing effective memory scheduling policies in the presence of these two different types of applications/access-patterns.)

6. New Research Challenge 3: Predictable Performance

Since memory is a shared resource between multiple cores (or, agents, threads, or applications and virtual machines), as shown in Figure 8, different applications contend for bandwidth and capacity at the different components of the memory system, such as memory controllers, interconnects and caches. As such, memory contention, or memory interference, between different cores critically affects both the overall system performance and each application's performance. Our past work (e.g., [129, 137, 138, 142]) showed that application-unaware design of memory controllers, and in particular, memory scheduling algorithms, leads to uncontrolled interference of applications in the memory system. Such uncontrolled interference can lead to denial of service to some applications [129], low system performance [137, 138, 142], unfair and unpredictable application slowdowns [53, 56, 137, 176]. For instance, Figure 9 shows the application slowdowns when two applications are run together on a simulated two-core system where the two cores share the main memory (including the memory controllers). The application leslie3d (from the SPEC CPU2006 suite) slows down significantly due to interference from the co-running application. Furthermore, leslie3d's slowdown depends heavily on the co-running application. It slows down by 2x when run with gcc, whereas it slows down by more than 5x when run with mcf, an application that exercises the memory significantly. Our past works have shown that similarly unpredictable and uncontrollable slowdowns happen in both existing systems (e.g., [88, 129]) and simulated systems (e.g., [53, 56, 137, 138, 142, 176]), across a wide variety of workloads

Building QoS and application awareness into the different components of the memory system such as memory controllers, caches, interconnects is important to control interference at these different components and mitigate/eliminate unfairness and unpredictability. Towards this end, previous works have explored two different solution directions: 1) to mitigate interference, thereby reducing application slowdowns and improving overall system performance, 2) to precisely quantify and control the impact of interference on application slowdowns, thereby providing performance guarantees to applications that need such guarantees. We discuss these two different approaches and associated research problems next.

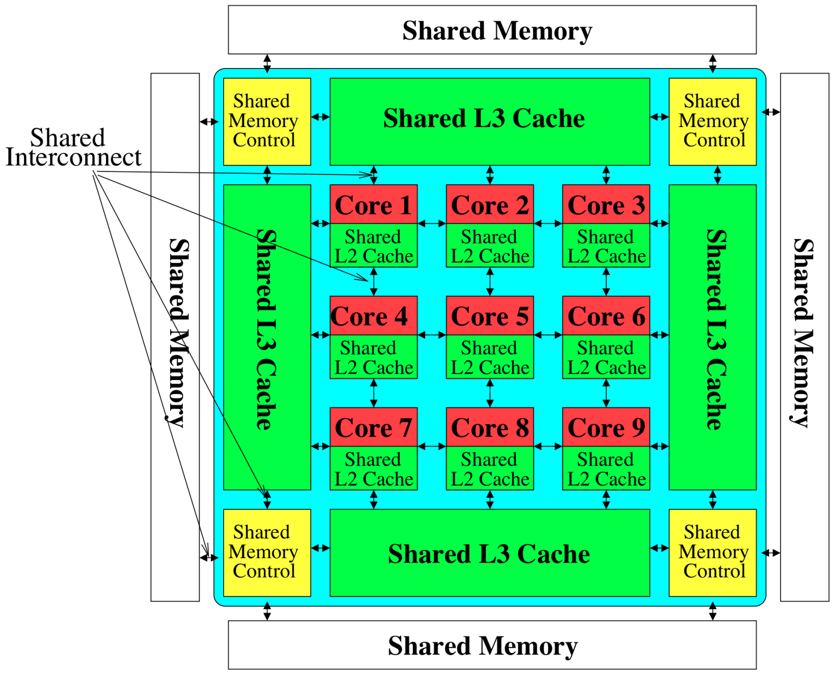
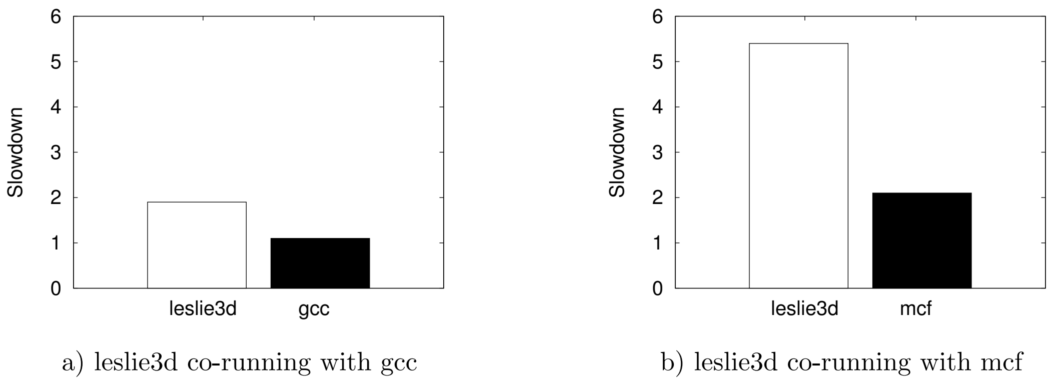


Figure 8. A typical multicore system. Reproduced from [133]



a) leslie3d co-running with gcc b) leslie3d co-running with mcf

Figure 9. High and unpredictable application slowdowns

6.1. Mitigating Interference

In order to mitigate interference at the different components of memory system, two kinds of approaches have been explored. The resources could be either *smart* (i.e., aware of threads' or applications' interference in memory) or *dumb* (i.e., unaware of threads' or applications' interference in memory) as we describe below.[3]

6.1.1. Smart Resources

The *smart resources* approach equips resources (such as memory controllers, interconnects and caches) with the intelligence to 1) be aware of interference characteristics between applications and 2) prevent unfair application slowdowns and performance degradation. Several of our past works have taken the *smart resources* approach and designed QoS-aware memory controllers [11, 54, 88, 93–95, 105, 130, 137, 138, 142, 176, 177] and inter-

[3]For the rest of this article, without loss of generality, we use the terms *thread* and *application* interchangeably. From the point of view of a *smart* resource, the resource needs to be communicated at least the *hardware context identifier* of the application/thread that is generating requests to be serviced.

connects [27, 41–43, 64, 66, 67, 128, 147, 148]. Our and other previous works have explored smart shared cache management policies that 1) allocate shared cache capacity to applications in a manner that is aware of their cache utility [161, 195], 2) modify the cache replacement and insertion policies to be aware of the data reuse and memory access behavior of applications [75, 76, 87, 160, 166, 169]. All these QoS-aware schemes enable resources (such as memory controllers, interconnects and caches) to detect interference between applications by means of monitoring their access characteristics and allocate resources such as memory bandwidth, interconnect link bandwidth and cache capacity to applications so that interference between applications is mitigated.

We provide several brief examples of the *smart resources* approach by focusing on QoS-aware memory controllers [11, 54, 88, 93–95, 105, 130, 137, 138, 142, 176, 177].

Mutlu and Moscibroda [129, 137] devised some of the first fair memory controllers. Their memory scheduler dynamically estimates the slowdown of each application and prioritizes applications' requests in a way that balances the slowdowns. In a later work, Mutlu and Moscibroda [138, 142] show that uncontrolled inter-thread interference in the memory controller can destroy the memory-level parallelism (MLP) [35, 51, 63, 140, 141, 143, 144, 150] and serialize requests of individual threads, leading to significant degradation in both single-thread and system performance in multi-core/multi-threaded systems. Hence, many techniques devised by computer architects to parallelize a thread's memory requests to tolerate memory latency by exploiting MLP, such as out-of-order execution [151, 152, 185], non-blocking caches [99], runahead execution [30, 35, 51, 136, 139–141, 143, 144] and other techniques [34, 160, 205], can become ineffective if the memory controller is not aware of threads. To overcome this and ensure the memory controller can serve each thread's requests in parallel, they introduce the idea of *thread ranking*, where a memory controller forms a rank order among threads and services threads in that order. To provide high fairness and starvation freedom, their controller employs *batching* of requests, where the memory controller groups oldest requests from each thread into a batch and services that batch before all other requests. This work, and the associated memory scheduler PAR-BS (Parallelism-Aware Batch Scheduler) has formed the basis of many future thread-aware memory scheduling policies by providing relatively simple and effective mechanisms for *both* performance and fairness in the memory controller.

Kim et al. [93] observe that applications that have received the least service from the memory controllers in the past, would benefit from quick request service (in the future) and hence, seek to prioritize such applications' requests at the memory controller, with the goal of improving system performance. They propose ATLAS [93], a QoS-aware memory controller design that monitors and ranks applications based on the amount of service each application receives at the memory controllers and prioritizes requests of applications with low *attained memory service*. ATLAS provides significant performance improvement by prioritizing applications that can benefit the most from memory service. In a later work, Kim et al. [94, 95], observe that prioritizing between *all* applications *solely based on one access characteristic*, e.g., the attained memory service in ATLAS, could unfairly slow down some applications: applications that are very memory intensive can be slowed down disproportionately. To solve this problem, they propose the thread cluster memory scheduler (TCM) [94, 95], which employs heterogeneous request scheduling policies for different types of applications: latency-sensitive vs. bandwidth-sensitive, as shown in Figure 10. TCM classifies applications into two clusters, the low- and high-memory-intensity (or, latency-sensitive and bandwidth-sensitive) clusters, prioritizes the latency-sensitive cluster, and employs a different policy to rank and prioritize among applications within each cluster, with the goal of optimizing for both performance and fairness. Results show that TCM can achieve both high performance and fairness compared to the best schedulers of the time.

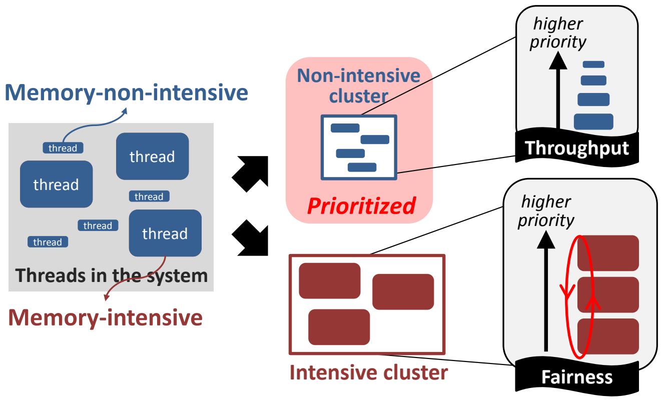


Figure 10. Operation of Thread Cluster Memory Scheduler (TCM). Reproduced from [92]

Most recently, Subramanian et al. [177] propose the blacklisting memory scheduler (BLISS) based on the observation that it is *not* necessary to employ a *full ordered ranking* across all applications, like PAR-BS, ATLAS and TCM do, because of two key reasons. First, such a full ordered ranking scheme incurs *high hardware complexity*. Second, a full ordered ranking of applications prioritizes some high-memory-intensity applications over other high-memory-intensity applications, resulting in *unfair application slowdowns*. Hence, they propose to separate applications into *only two groups* (instead of employing a full ranking of threads), one containing interference-causing applications and the other containing vulnerable-to-interference applications and prioritize the vulnerable-to-interference group over the interference-causing group. These groups are formed by monitoring the number of consecutive requests from an applications and classifying applications that generate more than a certain number of consecutive requests as interference-causing (this is called *blacklisting*). Such a scheme not only greatly reduces the hardware complexity and critical path latency of the memory scheduler (as it *does not require full ranking of all applications*), but also prevents applications from slowing down unfairly, thereby improving system performance.

It is worth noting that inter-thread interference in the memory controller among threads of the *same* application can greatly reduce that application's performance as well. Ebrahimi et al. [54] quantify the performance loss due to such interference and propose a new memory controller that dynamically estimates *critical threads* (or, *limiter threads*) in an application, which limit performance, and prioritizes such threads over others. Such an approach that identifies the most important threads, potentially using various other mechanisms [14, 18, 50, 78, 79, 178–181], and prioritizes/accelerates them in not only the memory controllers but also other resources is likely to be promising to improve parallel program performance and efficiency.

The design of memory controllers remains equally, if not more, important in the presence of persistent memory systems that store and access persistent data through the memory interface (as we discussed in Section 5.3). In fact, in such systems *memory writes* can become very frequent as persistent data needs to be flushed to main memory in a strict order determined by the storage consistency model employed in modern systems [204]. Zhao et al. [204] identified this problem and showed that existing memory controllers cannot appropriately handle interference between applications that access persistent data and applications that access volatile data because those that write to persistent data can greatly reduce system performance and fairness due to scheduling policies that do not take into account write requests. They develop a new memory scheduling algorithm that provides a solution to this problem by more fairly handling read and write requests of different applications [204].

Finally, it is critically important to appropriately handle the interference caused by *prefetch requests* generated by hardware and software prefetchers employed in almost all modern high performance systems [69, 174, 183]. Lee et al. [105, 107, 108] show that making the memory controller dynamically decide between providing equal or lower priority to prefetch requests compared to demand requests, based on the accuracy of prefetches, can greatly improve system performance and fairness. Ebrahimi et al. [55, 57, 58] showed that interference caused by aggressive prefetchers, even if they are accurate, can cause slowdowns to applications and reduce performance. They devise mechanisms to appropriately throttle prefetchers to reduce the negative effects of the interference caused.

All these past works on QoS-aware and interference-aware memory scheduling have shown that significant performance and fairness gains are possible by designing the memory controller to be aware of different threads/applications and different request/access characteristics and appropriately prioritizing among them. We believe ample opportunity exists for future designs that can effectively navigate the complex tradeoff space of performance, fairness, hardware complexity/cost, scheduling latency and energy efficiency. Designs that optimize for three or more of these metrics at the same time, e.g., in the spirit of BLISS [177], will especially be desirable in the future and are a promising direction for future research.

A challenge with the *smart resources* approach is the coordination of resource allocation decisions across different resources, such as main memory, interconnects and shared caches. To provide QoS across the entire system, the actions of different resources need to be coordinated. Several works examined such coordination mechanisms. One example of such a scheme is a coordinated resource management scheme proposed by Bitirgen et al. [15] that employs machine learning, specifically, an artificial neural network, to predict each application's performance for different possible resource allocations. Resources are then allocated appropriately to different applications so that a global system performance metric is optimized. Another example of such a scheme is a recent work by Wang and Martinez [191] that employs a market-dynamics-inspired mechanism to coordinate allocation decisions across resources. Approaches to coordinate resource allocation and scheduling decisions across multiple resources in the memory system, whether they use machine learning, game theory, or feedback based control, are a promising research topic that offers ample scope for future exploration.

6.1.2. Dumb Resources

The *dumb resources* approach, rather than modifying the resources themselves to be QoS- and application-aware, controls the resources and their allocation at different points in the system (e.g., at the cores/sources or at the system software) so that unfair slowdowns and performance degradation are mitigated. For instance, Ebrahimi et al. propose Fairness via Source Throttling (FST) [53, 55, 56], which throttles applications at the source (processor core) to regulate the number of requests that are sent to the shared caches and main memory from the processor core. Cheng et al. [32] propose to break down threads into compute and memory tasks and restrict the number of concurrent memory tasks. Kayiran et al. [85] throttle the thread-level parallelism of the GPU to mitigate memory contention related slowdowns in heterogeneous architectures consisting of both CPUs and GPUs. Das et al. [40] propose to map applications to cores (by modifying the application scheduler in the operating system) in a manner that is aware of the applications' interconnect and memory access characteristics. Muralidhara et al. [131] propose to partition memory channels among applications such the data of applications that interfere significantly with each other are mapped to different memory channels. Other works [77, 88, 116, 194] build upon [131] and partition banks between applications at a fine-grained level to prevent different applications' request streams from interfering at the same banks. Zhuravlev et al. [206] and Tang et al. [182] propose to mitigate interference by co-scheduling threads that interact well and interfere less at the shared resources. Kaseridis et al. [84] propose to interleave data across channels and banks such that threads benefit from locality in the DRAM row-buffer, while not causing significant interference to each other.

On the interconnect side, there has been a growing body of work on *source throttling* mechanisms [12, 27, 61, 65, 147, 148, 184] that detect congestion or slowdowns within the network and throttle the injection rates of applications in an application-aware manner to maximize both system performance and fairness. In heterogeneous architectures consisting of CPUs and GPUs, similar *source throttling* approaches can greatly reduce memory congestion and improve both QoS provided to the CPUs and overall system performance [85]. Kayiran et al. [85] provide a promising mechanism that can be configured to achieve multiple different performance goals in such architectures by adapting the number of threads scheduled in the GPU based on memory congestion and latency tolerance characteristics in the heterogeneous system.

One common characteristic among all these *dumb resources* approaches is that they regulate the amount of contention at the caches, interconnects and main memory by controlling their operation/allocation from a different agent such as the cores, the operating system, or the memory allocator, while *not* modifying the resources themselves to be QoS- or application-aware. This has the advantage of keeping the resources themselves simple, while also potentially enabling better coordination of allocation decisions across multiple resources. On the other hand, the disadvantages of the *dumb resources* approach are that 1) each resource may not be best utilized to provide the highest performance because it cannot make interference-aware decisions, 2) monitoring and control mechanisms to understand and decide how to best control operation/allocation from a different agent than the resources themselves to achieve a particular goal increase complexity.

6.1.3. Integrated Approaches to QoS and Resource Management

Even though both the *smart resources* and *dumb resources* approaches can greatly mitigate interference and provide QoS in the memory system when employed individually, they each have advantages and disadvantages that are unique, as we discussed above. Therefore, integrating the *smart resources* and *dumb resources* approaches can enable better interference mitigation than employing either approach alone by exploiting the advantages of each approach. An early example of such an integrated resource management approach is the Integrated Memory Partitioning and Scheduling (IMPS) scheme [131], which combines memory channel partitioning in the operating system (a *dumb resource* approach) along with simple application-aware memory request scheduling in the memory controller (a *smart resource* approach), leading to higher performance than when either approach is employed alone. The key idea of IMPS is to partition memory channels to mitigate interference between memory-intensive applications while prioritizing compute-intensive applications in the memory scheduler. Subramanian et al. [131] show that this combined technique improves performance more than either memory channel partitioning or application-aware memory scheduling alone. We believe such combined approaches will become even more important in the future as memory interference becomes an even more severe problem than today due to limited memory bandwidth and data-intensive workloads. Combining different approaches to memory QoS and resource management, both *smart* and *dumb*, with the goal of more effective interference mitigation is therefore a promising area for future research and exploration.

6.2. Quantifying and Controlling Interference

While several previous works have focused on mitigating interference at the different components of a memory system, with the goals of improving performance and preventing unfair application slowdowns, few previous works have focused on *precisely* quantifying and controlling the impact of interference on application slowdowns, with the goal of providing *soft or hard performance guarantees*. An application's susceptibility to interference and, consequently, its performance, depends on which other applications it is sharing resources with: an application can sometimes have very high performance and at other times very low performance on the same system, solely depending on its co-runners (as we have already discussed in Section 6 and shown an example in Figure 9). Therfore a critical research challenge is how to design the memory system (including all shared resources such as main memory, caches, and interconnects) so that 1) the performance of *each application* is predictable and controllable, and performance requirements of each application are satisfied, while 2) the performance and efficiency of the *entire system* are as high as needed or possible.

A promising solution direction to address this *predictable performance* challenge is to devise mechanisms that are effective and accurate at 1) estimating and predicting application performance in the presence of inter-application interference in a dynamic system with continuously incoming and outgoing applications, and 2) enforcing *end-to-end performance guarantees* within the entire shared memory system. We discuss briefly several prior works that took some of the first steps to achieve these goals and conclude with some promising future directions to pursue.

Stall Time Fair Memory Scheduling (STFM) [137] is one of the first works on estimating the impact of inter-application memory interference on application performance. STFM estimates the impact of memory interference at the main memory alone on application slowdowns. It does so by estimating the impact of delaying every individual request of an application on its slowdown. Fairness via Source Throttling (FST) [53, 56] and Per-Thread Cycle Accounting [49] employ a scheme similar to STFM to estimate the impact of main memory interference, while also taking into account interference at the shared caches. While these approaches are good starting points towards addressing the challenge of estimating and predicting application slowdowns in the presence of memory interference, our recent work [176] observed that the slowdown estimates from STFM, FST and Per-Thread Cycle Accounting are relatively inaccurate since they estimate interference at an *individual request granularity*. As a result, such schemes may not only be able to provide strict performance guarantees but also become complex due to the hardware needed to track interference on an individual request granularity.

We have recently designed a simple method, called MISE (Memory-interference Induced Slowdown Estimation) [176], for estimating application slowdowns accurately in the presence of main memory interference. We observe that an application's aggregate memory request service rate is a good proxy for its performance, as depicted in Figure 11, which shows the measured performance versus memory request service rate for three applications on a real system [176]. As such, an application's slowdown can be accurately estimated by estimating its *uninterfered request service rate*, which can be done by prioritizing that application's requests in the memory system during some execution intervals. Results show that average error in slowdown estimation with this relatively simple technique is approximately 8% across a wide variety of workloads. Figure 12 shows the actual versus predicted slowdowns over time, for astar, a representative application from among the many applications examined, when it is run alongside three other applications on a simulated 4-core system. As we can see, MISE's slowdown estimates track the actual measured slowdown closely.

We believe these works on accurately estimating application slowdowns and providing predictable performance in the presence of memory interference have just scratched the surface of a critically important research direction. Designing predictable computing systems is a Grand Research Challenge, as identified by the Computing Research Association [37]. Many future ideas in this direction seem promising. We discuss some of them briefly. First, extending such simple performance estimation techniques like MISE [176] to the entire memory and storage system is a promising area of future research in both homogeneous and heterogeneous systems. Second, estimating and bounding memory slowdowns for hard real-time performance guarantees, as recently discussed by Kim et al. [88], is similarly promising. Third, devising memory devices, architectures and interfaces that can support better predictability and QoS also appears promising. Some key exciting research questions include, but are by no means limited to, the following: How sensitive are different applications to memory, interconnect and storage bandwidth? How does this sensitivity vary with the memory/storage/interconnect technology? How sensitive are applications to memory/cache/storage capacity? How do we take into account sensitivity to different resources such as cache/memory/storage capacity and bandwidth in estimating application slowdowns? How can we estimate slowdowns in heterogeneous systems consisting of CPUs, GPUs and hardware accelerators?

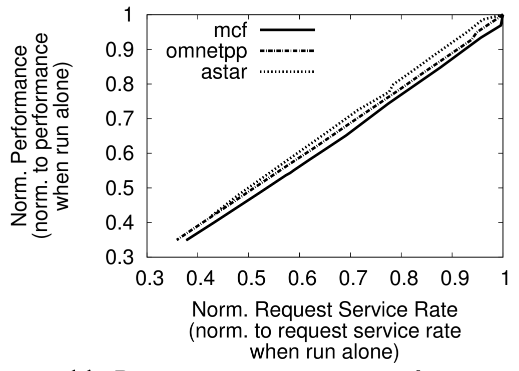


Figure 11. Request service rate vs. performance. Reproduced from [176]

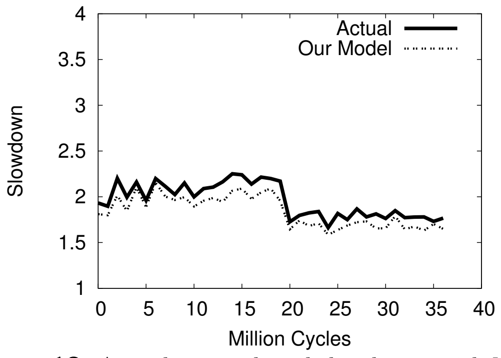


Figure 12. Actual vs. predicted slowdowns with MISE. Reproduced from [176]

Once accurate slowdown estimates are available, they can be leveraged in multiple possible ways. One possible use case is to leverage them in the hardware to allocate *just enough resources* to an application so that its performance requirements are met. We demonstrate such a scheme for memory bandwidth allocation showing that applications' performance/slowdown requirements can be effectively met by leveraging slowdown estimates from the MISE model [176]. There are several other ways in which slowdown estimates can be leveraged in both the hardware and the software to achieve various system-level goals. For instance, accurate slowdown estimates can be used to drive fair pricing schemes based on slowdowns, rather than just resource allocation, in a cloud computing setting [1, 2]. Slowdown estimates can also be used to consolidate virtual machines onto physical hosts so that applications are not unfairly slowed down, through virtual machine migration and admission control schemes [68, 117, 182]. These and other potential schemes to leverage accurate slowdown estimates are promising directions to explore.

7. Flash Memory Scaling Challenges

We discuss briefly the challenges of scaling the major component of the storage subsystem, flash memory. Flash memory is arguably the most successful charge-based memory (like DRAM) that has been employed as part of the storage system. Its benefits over hard disks are clear: greatly lower latencies, greatly higher bandwidth and much higher reliability due to the lack of mechanical components. These have lead to the successful adoption of flash memory in modern systems across the board, to the point of replacing hard disk drives completely in space-constrained environments, e.g., laptop computers.

Our discussion in this section will be limited to some key challenges in improving the reliability and lifetime of flash memory, which we have been exploring in our recent research. Note that many other challenges exist, including but not limited to the following: 1) making the storage stack much higher performance to take advantage of the low latency and high parallelism of raw flash devices [189] (similarly to what we discussed in Section 5.3 with respect to the Persistent Memory Manager), 2) providing transactional support in the flash translation layer for better system performance and flexibility [120], 3) taking advantage of application- and system-level information to manage flash memory in a way that improves performance, efficiency, lifetime and cost. These are great research directions to explore, but, for brevity, we will not discuss them in further detail.

In part of our research, we aim to develop new techniques that overcome reliability and endurance challenges of flash memory to enable its scaling beyond the 20nm technology generations. To this end, we experimentally measure, characterize, analyze, and model error patterns that occur in existing flash chips, using an experimental flash memory testing and characterization platform [19]. Based on the understanding we develop from our experiments, we aim to develop error management techniques that mitigate the fundamental types of errors that are likely to increase as flash memory scales.

We have recently experimentally characterized complex flash errors that occur at 30-40nm flash technologies [20], categorizing them into four types: retention errors, program interference errors, read errors, and erase errors. Our characterization shows the relationship between various types of errors and demonstrates empirically using real 3x-nm flash chips that retention errors are the most dominant error type. Our results demonstrate that different flash errors have distinct patterns: retention errors and program interference errors are program/erase-(P/E)-cycle-dependent, memory-location-dependent, and data-value-dependent. Since the observed error patterns are due to fundamental circuit and device behavior inherent in flash memory, we expect our observations and error patterns to also hold in flash memories beyond 30-nm technology.

Based on our experimental characterization results that show that the retention errors are the most dominant errors, we have developed a suite of techniques to mitigate the effects of such errors, called Flash Correct-and-Refresh (FCR) [21]. The key idea is to periodically read each page in flash memory, correct its errors using simple error correcting codes (ECC), and either remap (copy/move) the page to a different location or reprogram it in its original location by recharging the floating gates, before the page accumulates more errors than can be corrected with simple ECC. Our simulation experiments using real I/O workload traces from a variety of file system, database, and search applications show that FCR can provide 46x flash memory lifetime improvement at only 1.5% energy overhead, with no additional hardware cost.

We have also experimentally investigated and characterized the threshold voltage distribution of different logical states in MLC NAND flash memory [24]. We have developed new models that can predict the shifts in the threshold voltage distribution based on the number of P/E cycles endured by flash memory cells. Our data shows that the threshold voltage distribution of flash cells that store the same value can be approximated, with reasonable accuracy, as a Gaussian distribution. The threshold voltage distribution of flash cells that store the same value gets distorted as the number of P/E cycles increases, causing threshold voltages of cells storing different values to overlap with each other, which can lead to the incorrect reading of values of some cells as flash cells accumulate P/E cycles. We find that this distortion can be accurately modeled and predicted as an exponential function of the P/E cycles, with more than 95% accuracy. Such predictive models can aid the design of more sophisticated error correction methods, such as LDPC codes [62], which are likely needed for reliable operation of future flash memories.

We are currently investigating another increasingly significant obstacle to MLC NAND flash scaling, which is the increasing cell-to-cell program interference due to increasing parasitic capacitances between the cells' floating gates. Accurate characterization and modeling of this phenomenon are needed to find effective techniques to combat program interference. In recent work [23], we leverage the *read retry* mechanism found in some flash designs to obtain measured threshold voltage distributions from state-of-the-art 2Y-nm (i.e., 24-20 nm) MLC NAND flash chips. These results are then used to characterize the cell-to-cell program interference under various programming conditions. We show that program interference can be accurately modeled as additive noise following Gaussian-mixture distributions, which can be predicted with 96.8% accuracy using linear regression models. We use these models to develop and evaluate a read reference voltage prediction technique that reduces the raw flash bit error rate by 64% and increases the flash lifetime by 30%. More details can be found in Cai et al. [23].

To improve flash memory lifetime, we have developed a mechanism called Neighbor-Cell Assisted Correction (NAC) [25], which uses the value information of cells in a neighboring page to correct errors found on a page when reading. This mechanism takes advantage of the new empirical observation that identifying the value stored in the immediate-neighbor cell makes it easier to determine the data value stored in the cell that is being read. The key idea is to re-read a flash memory page that fails error correction codes (ECC) with the set of read reference voltage values corresponding to the conditional threshold voltage distribution assuming a neighbor cell value and use the re-read values to correct the cells that have neighbors with that value. Our simulations show that NAC effectively improves flash memory lifetime by 33% while having no (at nominal lifetime) or very modest (less than 5% at extended lifetime) performance overhead.

Most recently, we have provided a rigorous characterization of retention errors in NAND flash memory devices and new techniques to take advantage of this characterization to improve flash memory lifetime [26]. We have extensively characterized how the threshold voltage distribution of flash memory changes under different retention age, i.e., the length of time since a flash cell is programmed. We observe from our characterization results that 1) the optimal read reference voltage of a flash cell, at which the data can be read with the lowest raw bit error rate (RBER), systematically changes with the cell's retention age and 2) different regions of flash memory can have different retention ages, and hence different optimal read reference voltages. Based on these observations, we propose a new technique to learn and apply the optimal read reference voltage online (called retention optimized reading). Our evaluations show that our technique can extend flash memory lifetime by 64% and reduce average error correction latency by 7% with only a 768 KB storage overhead in flash memory for a 512 GB flash-based SSD. We also propose a technique to recover data with uncorrectable errors by identifying and *probabilistically correcting* flash cells with retention errors. Our evaluation shows that this technique can effectively recover data from uncorrectable flash errors and reduce RBER by 50%. More detail can be found in Cai et al. [26].

These works, to our knowledge, are the first open-literature works that 1) characterize various aspects of real state-of-the-art flash memory chips, focusing on reliability and scaling challenges, and 2) exploit the insights developed from these characterizations to develop new mechanisms that can improve flash memory reliability and lifetime. We believe such an experimental- and characterization-based approach (which we also employ for DRAM [86, 97, 110, 115], as we discussed in Section 4) to developing novel techniques for both existing and emerging memory technologies is critically important as it 1) provides a solid basis (i.e., real data from modern devices) on which future analyses and new techniques can be based, 2) reveals new scaling trends in modern devices, pointing to important challenges in the field, and 3) equips the research community with reliable models and analyses that can be openly used to innovate in areas where experimental information is usually scarce in the open literature due to heavy competition within industry, thereby enhancing research and investigation in areas that are previously deemed to be difficult to research.

Going forward, we believe more accurate and detailed characterization of flash memory error mechanisms is needed to devise models that can aid the design of even more efficient and effective mechanisms to tolerate errors found in sub-20nm flash memories. A promising direction is the design of predictive models that the system (e.g., the flash controller or system software) can use to proactively estimate the occurrence of errors and take action to prevent the error before it happens. Flash-correct-and-refresh [21], read reference voltage prediction [23], and retention optimized reading [26] mechanisms, described earlier, are early forms of such predictive error tolerance mechanisms. Methods for exploiting application and memory access characteristics to optimize flash performance, energy, lifetime and cost are also very promising to explore. We believe there is a bright future ahead for more aggressive and effective application- and data-characteristic-aware management mechanisms for flash memory (just like for DRAM and emerging memory technologies). Such techniques will likely aid effective scaling of flash memory technology into the future.

8. Conclusion

We have described several research directions and ideas to enhance memory scaling via system and architecture-level approaches. We believe there are three key *fundamental principles* that are essential for memory scaling: 1) better cooperation between devices, system, and software, i.e., the efficient exposure of richer information up and down the layers of the system stack with the development of more flexible yet abstract interfaces that can scale well into the future , 2) better-than-worst-case design, i.e., design of the memory system such that it is optimized for the common case instead of the worst case, 3) heterogeneity in design, i.e., the use of heterogeneity at all levels in memory system design to enable the optimization of multiple metrics at the same time. We believe these three principles are related and sometimes coupled. For example, to exploit heterogeneity in the memory system, we may need to enable better cooperation between the devices and the system, e.g., as in the case of heterogeneous DRAM refresh rates [114], tiered-latency DRAM [109], heterogeneous-reliability memory [122], locality-aware management of hybrid memory systems [201] and the persistent memory manager for a heterogeneous array of memory/storage devices [127], five of the many ideas we have discussed in this paper.

We have shown that a promising approach to designing scalable memory systems is the co-design of memory and other system components to enable better system optimization. Enabling better cooperation across multiple levels of the computing stack, including software, microarchitecture, and devices can help scale the memory system by exposing more of the memory device characteristics to higher levels of the system stack such that the latter can tolerate and exploit such characteristics. Finally, heterogeneity in the design of the memory system can help overcome the memory scaling challenges at the device level by enabling better specialization of the memory system and its dynamic adaptation to different demands of various applications. We believe such system-level and integrated approaches will become increasingly important and effective as the underlying memory technology nears its scaling limits at the physical level and envision a near future full of innovation in main memory architecture, enabled by the co-design of the system and main memory.

Acknowledgments

The source code and data sets of some of the works we have discussed or alluded to (e.g., [9, 86, 97, 111, 153, 166, 196]) are available under open source software license at our research group, SAFARI's, tools website [3].

We would like to thank Rachata Ausavarungnirun for logistic help in preparing this article and earlier versions of it. Many thanks to all students in the SAFARI research group and collaborators at Carnegie Mellon as well as other universities, whom have contributed to the various works outlined in this article. Thanks also go to our research group's industrial sponsors over the past six years, including AMD, HP Labs, Huawei, IBM, Intel, Microsoft, Nvidia, Oracle, Qualcomm, Samsung. Some of the research reported here was also partially supported by GSRC, Intel URO Memory Hierarchy Program, Intel Science and Technology Center on Cloud Computing, NIH, NSF, and SRC. In particular, NSF grants 0953246, 1065112, 1147397, 1212962, 1320531, 1409723, 1423172 have supported parts of the research we have described.

This article is a significantly extended and revised version of an invited paper that appeared at the 5th International Memory Workshop [133], which was also presented at MemCon 2013 [134]. Part of the structure of this article is based on an evolving set of talks Onur Mutlu has delivered at various venues on *Scaling the Memory System in the Many-Core Era* and *Rethinking Memory System Design for Data-Intensive Computing* between 2010-2015, including invited talks at the 2011 International Symposium on Memory Management and ACM SIGPLAN Workshop on Memory System Performance and Correctness [135] and the 2012 DAC More Than Moore Technologies Workshop. Section 7 of this article is a significantly extended and revised version of the introduction of an invited article that appeared in a special issue of the Intel Technology Journal, titled *Error Analysis and Retention-Aware Error Management for NAND Flash Memory* [22].

References

1. *Amazon EC2*, http://aws.amazon.com/ec2/pricing/.
2. *Microsoft Azure*, http://azure.microsoft.com/en-us/pricing/details/virtual-machines/.
3. “SAFARI tools,” https://www.ece.cmu.edu/~safari/tools.html.
4. “International technology roadmap for semiconductors (ITRS),” 2011.
5. *Hybrid Memory Consortium*, 2012, http://www.hybridmemorycube.org.
6. *Top 500*, 2013, http://www.top500.org/featured/systems/tianhe-2/.
7. J.-H. Ahn *et al.*, “Adaptive self refresh scheme for battery operated high-density mobile DRAM applications,” in *ASSCC*, 2006.
8. A. R. Alameldeen and D. A. Wood, “Adaptive cache compression for high-performance processors,” in *ISCA*, 2004.
9. C. Alkan *et al.*, “Personalized copy-number and segmental duplication maps using next-generation sequencing,” in *Nature Genetics*, 2009.
10. G. Atwood, “Current and emerging memory technology landscape,” in *Flash Memory Summit*, 2011.
11. R. Ausavarungnirun *et al.*, “Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems,” in *ISCA*, 2012.
12. R. Ausavarungnirun *et al.*, “Design and evaluation of hierarchical rings with deflection routing,” in *SBAC-PAD*, 2014.
13. S. Balakrishnan and G. S. Sohi, “Exploiting value locality in physical register files,” in *MICRO*, 2003.
14. A. Bhattacharjee and M. Martonosi, “Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors,” in *ISCA*, 2009.
15. R. Bitirgen *et al.*, “Coordinated management of multiple interacting resources in CMPs: A machine learning approach,” in *MICRO*, 2008.
16. B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, 1970.
17. R. Bryant, “Data-intensive supercomputing: The case for DISC,” *CMU CS Tech. Report 07-128*, 2007.
18. Q. Cai *et al.*, “Meeting points: Using thread criticality to adapt multicore hardware to parallel regions,” in *PACT*, 2008.
19. Y. Cai *et al.*, “FPGA-based solid-state drive prototyping platform,” in *FCCM*, 2011.
20. Y. Cai *et al.*, “Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis,” in *DATE*, 2012.
21. Y. Cai *et al.*, “Flash Correct-and-Refresh: Retention-aware error management for increased flash memory lifetime,” in *ICCD*, 2012.

22. Y. Cai *et al.*, "Error analysis and retention-aware error management for NAND flash memory," *Intel technology Journal*, vol. 17, no. 1, May 2013.
23. Y. Cai *et al.*, "Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation," in *ICCD*, 2013.
24. Y. Cai *et al.*, "Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis and modeling," in *DATE*, 2013.
25. Y. Cai *et al.*, "Neighbor-cell assisted error correction for MLC NAND flash memories," in *SIGMETRICS*, 2014.
26. Y. Cai *et al.*, "Data retention in MLC NAND flash memory: Characterization, optimization and recovery," in *HPCA*, 2015.
27. K. Chang *et al.*, "HAT: Heterogeneous adaptive throttling for on-chip networks," in *SBAC-PAD*, 2012.
28. K. Chang *et al.*, "Improving DRAM performance by parallelizing refreshes with accesses," in *HPCA*, 2014.
29. N. Chatterjee *et al.*, "Leveraging heterogeneity in DRAM main memories to accelerate critical word access," in *MICRO*, 2012.
30. S. Chaudhry *et al.*, "High-performance throughput computing," *IEEE Micro*, vol. 25, no. 6, 2005.
31. E. Chen *et al.*, "Advances and future prospects of spin-transfer torque random access memory," *IEEE Transactions on Magnetics*, vol. 46, no. 6, 2010.
32. H.-Y. Cheng *et al.*, "Memory latency reduction via thread throttling," in *MICRO*, 2010.
33. S. Chhabra and Y. Solihin, "i-NVMM: a secure non-volatile main memory system with incremental encryption," in *ISCA*, 2011.
34. Y. Chou *et al.*, "Store memory-level parallelism optimizations for commercial applications," in *MICRO*, 2005.
35. Y. Chou *et al.*, "Microarchitecture optimizations for exploiting memory-level parallelism," in *ISCA*, 2004.
36. E. Chung *et al.*, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs?" in *MICRO*, 2010.
37. *Grand Research Challenges in Information Systems*, Computing Research Association, http://www.cra.org/reports/gc.systems.pdf.
38. J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*, 2009.
39. K. V. Craeynest *et al.*, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ISCA*, 2012.
40. R. Das *et al.*, "Application-to-core mapping policies to reduce memory system interference in multi-core systems," in *HPCA*, 2013.
41. R. Das *et al.*, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.
42. R. Das *et al.*, "Aergia: Exploiting packet latency slack in on-chip networks," in *ISCA*, 2010.
43. R. Das *et al.*, "Aergia: A network-on-chip exploiting packet latency slack," *IEEE Micro (TOP PICKS Issue)*, vol. 31, no. 1, 2011.

44. H. David *et al.*, “Memory power management via dynamic voltage/frequency scaling,” in *ICAC*, 2011.

45. J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, no. 2, 2013.

46. Q. Deng *et al.*, “MemScale: active low-power modes for main memory,” in *ASPLOS*, 2011.

47. G. Dhiman *et al.*, “PDRAM: A hybrid PRAM and DRAM main memory system,” in *DAC*, 2009.

48. X. Dong *et al.*, “Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems,” in *SC*, 2009.

49. K. Du Bois *et al.*, “Per-thread cycle accounting in multicore processors,” *TACO*, 2013.

50. K. Du Bois *et al.*, “Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior,” in *ISCA*, 2013.

51. J. Dundas and T. Mudge, “Improving data cache performance by pre-executing instructions under a cache miss,” in *ICS*, 1997.

52. J. Dusser *et al.*, “Zero-content augmented caches,” in *ICS*, 2009.

53. E. Ebrahimi *et al.*, “Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems,” in *ASPLOS*, 2010.

54. E. Ebrahimi *et al.*, “Parallel application memory scheduling,” in *MICRO*, 2011.

55. E. Ebrahimi *et al.*, “Prefetch-aware shared-resource management for multi-core systems,” in *ISCA*, 2011.

56. E. Ebrahimi *et al.*, “Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems,” *TOCS*, 2012.

57. E. Ebrahimi *et al.*, “Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems,” in *HPCA*, 2009.

58. E. Ebrahimi *et al.*, “Coordinated control of multiple prefetchers in multi-core systems,” in *MICRO*, 2009.

59. M. Ekman, “A robust main-memory compression scheme,” in *ISCA*, 2005.

60. S. Eyerman and L. Eeckhout, “Modeling critical sections in amdahl’s law and its implications for multicore design,” in *ISCA*, 2010.

61. C. Fallin *et al.*, “CHIPPER: a low-complexity bufferless deflection router,” in *HPCA*, 2011.

62. R. Gallager, “Low density parity check codes,” 1963, MIT Press.

63. A. Glew, “MLP yes! ILP no!” in *ASPLOS Wild and Crazy Idea Session*, Oct. 1998.

64. B. Grot *et al.*, “Kilo-NOC: A heterogeneous network-on-chip architecture for scalability and service guarantees,” in *ISCA*, 2011.

65. B. Grot *et al.*, “Regional congestion awareness for load balance in networks-on-chip,” in *HPCA*, 2008.

66. B. Grot *et al.*, “Preemptive virtual clock: A flexible, efficient, and cost-effective QOS scheme for networks-on-chip,” in *MICRO*, 2009.

67. B. Grot *et al.*, “Topology-aware quality-of-service support in highly integrated chip multiprocessors,” in *WIOSCA*, 2010.

68. A. Gulati *et al.*, "VMware distributed resource management: Design, implementation, and lessons learned," *VMware Technical Journal*, 2012.
69. G. Hinton *et al.*, "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, Feb. 2001, Q1 2001 Issue.
70. S. Hong, "Memory technology trend and future challenges," in *IEDM*, 2010.
71. E. Ipek *et al.*, "Self-optimizing memory controllers: A reinforcement learning approach," in *ISCA*, 2008.
72. C. Isen and L. K. John, "Eskimo: Energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *MICRO*, 2009.
73. R. Iyer, "CQoS: a framework for enabling QoS in shared caches of CMP platforms," in *ICS*, 2004.
74. R. Iyer *et al.*, "QoS policies and architecture for cache/memory in CMP platforms," in *SIGMETRICS*, 2007.
75. A. Jaleel *et al.*, "Adaptive insertion policies for managing shared caches," in *PACT*, 2008.
76. A. Jaleel *et al.*, "High performance cache replacement using re-reference interval prediction," in *ISCA*, 2010.
77. M. K. Jeong *et al.*, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *HPCA*, 2012.
78. J. A. Joao *et al.*, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*, 2012.
79. J. A. Joao *et al.*, "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *ISCA*, 2013.
80. A. Jog *et al.*, "Orchestrated scheduling and prefetching for GPGPUs," in *ISCA*, 2013.
81. A. Jog *et al.*, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," in *ASPLOS*, 2013.
82. T. L. Johnson *et al.*, "Run-time spatial locality detection and optimization," in *MICRO*, 1997.
83. U. Kang *et al.*, "Co-architecting controllers and DRAM to enhance DRAM process scaling," in *The Memory Forum*, 2014.
84. D. Kaseridis *et al.*, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *MICRO*, 2011.
85. O. Kayiran *et al.*, "Managing GPU concurrency in heterogeneous architectures," in *MICRO*, 2014.
86. S. Khan *et al.*, "The efficacy of error mitigation techniques for DRAM retention failures: A comparative experimental study," in *SIGMETRICS*, 2014.
87. S. Khan *et al.*, "Improving cache performance by exploiting read-write disparity," in *HPCA*, 2014.
88. H. Kim *et al.*, "Bounding memory interference delay in COTS-based multi-core systems," in *RTAS*, 2014.
89. J. Kim and M. C. Papaefthymiou, "Dynamic memory design for low data-retention power," in *PATMOS*, 2000.

90. K. Kim, "Future memory technology: challenges and opportunities," in *VLSI-TSA*, 2008.
91. K. Kim *et al.*, "A new investigation of data retention time in truly nanoscaled DRAMs." *IEEE Electron Device Letters*, vol. 30, no. 8, Aug. 2009.
92. Y. Kim, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Talk at MICRO*, 2010.
93. Y. Kim *et al.*, "ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.
94. Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *MICRO*, 2010.
95. Y. Kim *et al.*, "Thread cluster memory scheduling," *IEEE Micro (TOP PICKS Issue)*, vol. 31, no. 1, 2011.
96. Y. Kim *et al.*, "A case for subarray-level parallelism (SALP) in DRAM," in *ISCA*, 2012.
97. Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.
98. Y. Koh, "NAND flash scaling beyond 20nm," in *IMW*, 2009.
99. D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *ISCA*, 1981.
100. E. Kultursay *et al.*, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.
101. S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *ISCA*, 1998.
102. B. C. Lee *et al.*, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.
103. B. C. Lee *et al.*, "Phase change memory architecture and the quest for scalability," *Communications of the ACM*, vol. 53, no. 7, 2010.
104. B. C. Lee *et al.*, "Phase change technology and the future of main memory," *IEEE Micro (TOP PICKS Issue)*, vol. 30, no. 1, 2010.
105. C. J. Lee *et al.*, "Prefetch-aware DRAM controllers," in *MICRO*, 2008.
106. C. J. Lee *et al.*, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," HPS, UT-Austin, Tech. Rep. TR-HPS-2010-002, 2010.
107. C. J. Lee *et al.*, "Prefetch-aware memory controllers," *TC*, vol. 60, no. 10, 2011.
108. C. J. Lee *et al.*, "Improving memory bank-level parallelism in the presence of prefetching," in *MICRO*, 2009.
109. D. Lee *et al.*, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *HPCA*, 2013.
110. D. Lee *et al.*, "Adaptive-latency DRAM: Optimizing DRAM timing for the common-case," in *HPCA*, 2015.
111. D. Lee *et al.*, "Fast and accurate mapping of complete genomics reads," in *Methods*, 2014.
112. C. Lefurgy *et al.*, "Energy management for commercial servers," in *IEEE Computer*, 2003.
113. K. Lim *et al.*, "Disaggregated memory for expansion and sharing in blade servers," in *ISCA*, 2009.
114. J. Liu *et al.*, "RAIDR: Retention-aware intelligent DRAM refresh," in *ISCA*, 2012.

115. J. Liu *et al.*, "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms," in *ISCA*, 2013.

116. L. Liu *et al.*, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012.

117. M. Liu and T. Li, "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads," in *ISCA*, 2014.

118. S. Liu *et al.*, "Flikker: saving DRAM refresh-power through critical data partitioning," in *ASPLOS*, 2011.

119. G. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA*, 2008.

120. Y. Lu *et al.*, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *ICCD*, 2013.

121. Y. Lu *et al.*, "Loose-ordering consistency for persistent memory," in *ICCD*, 2014.

122. Y. Luo *et al.*, "Characterizing application memory error vulnerability to optimize data center cost via heterogeneous-reliability memory," in *DSN*, 2014.

123. A. Maislos *et al.*, "A new era in embedded flash memory," in *FMS*, 2011.

124. J. Mandelman *et al.*, "Challenges and future directions for the scaling of dynamic random-access memory (DRAM)," in *IBM JR&D*, vol. 46, 2002.

125. J. Meza *et al.*, "A case for small row buffers in non-volatile main memories," in *ICCD*, 2012.

126. J. Meza *et al.*, "Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management," *IEEE CAL*, 2012.

127. J. Meza *et al.*, "A case for efficient hardware-software cooperative management of storage and memory," in *WEED*, 2013.

128. A. Mishra *et al.*, "A heterogeneous multiple network-on-chip design: An application-aware approach," in *DAC*, 2013.

129. T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security*, 2007.

130. T. Moscibroda and O. Mutlu, "Distributed order scheduling and its application to multi-core DRAM controllers," in *PODC*, 2008.

131. S. Muralidhara *et al.*, "Reducing memory interference in multi-core systems via application-aware memory channel partitioning," in *MICRO*, 2011.

132. O. Mutlu, "Asymmetry everywhere (with automatic resource management)," in *CRA Workshop on Advanced Computer Architecture Research*, 2010.

133. O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.

134. O. Mutlu, "Memory scaling: A systems architecture perspective," in *MemCon*, 2013.

135. O. Mutlu *et al.*, "Memory systems in the many-core era: Challenges, opportunities, and solution directions," in *ISMM*, 2011, http://users.ece.cmu.edu/~omutlu/pub/onur-ismm-mspc-keynote-june-5-2011-short.pptx.

136. O. Mutlu *et al.*, "Address-value delta (AVD) prediction: A hardware technique for efficiently parallelizing dependent cache misses," *IEEE Transactions on Computers*, vol. 55, no. 12, Dec. 2006.

137. O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

138. O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *ISCA*, 2008.

139. O. Mutlu *et al.*, "Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *MICRO*, 2005.

140. O. Mutlu *et al.*, "Techniques for efficient processing in runahead execution engines," in *ISCA*, 2005.

141. O. Mutlu *et al.*, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro (TOP PICKS Issue)*, vol. 26, no. 1, 2006.

142. O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enabling high-performance and fair memory controllers," *IEEE Micro (TOP PICKS Issue)*, vol. 29, no. 1, 2009.

143. O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA*, 2003.

144. O. Mutlu *et al.*, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro (TOP PICKS Issue)*, vol. 23, no. 6, 2003.

145. P. J. Nair *et al.*, "ArchShield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *ISCA*, 2013.

146. V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *MICRO*, 2011.

147. G. Nychis *et al.*, "Next generation on-chip networks: What kind of congestion control do we need?" in *HotNets*, 2010.

148. G. Nychis *et al.*, "On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects," in *SIGCOMM*, 2012.

149. T. Ohsawa *et al.*, "Optimizing the DRAM refresh count for merged DRAM/logic LSIs," in *ISLPED*, 1998.

150. V. S. Pai and S. Adve, "Code transformations to improve memory parallelism," in *MICRO*, 1999.

151. Y. N. Patt *et al.*, "HPS, a new microarchitecture: Rationale and introduction," in *MICRO*, 1985.

152. Y. N. Patt *et al.*, "Critical issues regarding HPS, a high performance microarchitecture," in *MICRO*, 1985.

153. G. Pekhimenko *et al.*, "Base-delta-immediate compression: A practical data compression mechanism for on-chip caches," in *PACT*, 2012.

154. G. Pekhimenko *et al.*, "Linearly compressed pages: A main memory compression framework with low complexity and low latency." in *MICRO*, 2013.

155. G. Pekhimenko *et al.*, "Exploiting compressed block size as an indicator of future reuse," in *HPCA*, 2015.

156. S. Phadke and S. Narayanasamy, "MLP aware heterogeneous memory system," in *DATE*, 2011.

157. M. K. Qureshi *et al.*, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *HPCA*, 2007.

158. M. K. Qureshi *et al.*, "Enhancing lifetime and security of phase change memories via start-gap wear leveling." in *MICRO*, 2009.

159. M. K. Qureshi *et al.*, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009.

160. M. K. Qureshi *et al.*, "A case for MLP-aware cache replacement," in *ISCA*, 2006.

161. M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006.

162. L. E. Ramos *et al.*, "Page placement in hybrid memory systems," in *ICS*, 2011.

163. S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM JR&D*, vol. 52, Jul/Sep 2008.

164. B. Schroder *et al.*, "DRAM errors in the wild: A large-scale field study," in *SIGMETRICS*, 2009.

165. N. H. Seong *et al.*, "Tri-level-cell phase change memory: Toward an efficient and reliable memory system," in *ISCA*, 2013.

166. V. Seshadri *et al.*, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *PACT*, 2012.

167. V. Seshadri *et al.*, "RowClone: Fast and efficient In-DRAM copy and initialization of bulk data," in *MICRO*, 2013.

168. V. Seshadri *et al.*, "The dirty-block index," in *ISCA*, 2014.

169. V. Seshadri *et al.*, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *TACO*, 2014.

170. F. Soltis, "Inside the AS/400," *29th Street Press*, 1996.

171. N. H. Song *et al.*, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *ISCA*, 2010.

172. V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC*, 2012.

173. V. Sridharan *et al.*, "Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *SC*, 2013.

174. S. Srinath *et al.*, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *HPCA*, 2007.

175. J. Stuecheli *et al.*, "The virtual write queue: Coordinating DRAM and last-level cache policies," in *ISCA*, 2010.

176. L. Subramanian *et al.*, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *HPCA*, 2013.

177. L. Subramanian *et al.*, "The blacklisting memory scheduler: Achieving high performance and fairness at low cost," in *ICCD*, 2014.

178. M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS*, 2009.

179. M. A. Suleman *et al.*, "Data marshaling for multi-core architectures," in *ISCA*, 2010.

180. M. A. Suleman *et al.*, "Data marshaling for multi-core systems," *IEEE Micro (TOP PICKS Issue)*, vol. 31, no. 1, 2011.

181. M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," *IEEE Micro (TOP PICKS Issue)*, vol. 30, no. 1, 2010.

182. L. Tang *et al.*, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011.

183. J. Tendler *et al.*, "POWER4 system microarchitecture," *IBM JRD*, Oct. 2001.

184. M. Thottethodi *et al.*, "Exploiting global knowledge to achieve self-tuned congestion control for k-ary n-cube networks," *IEEE TPDS*, vol. 15, no. 3, 2004.

185. R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM JR&D*, vol. 11, Jan. 1967.

186. T. Treangen and S. Salzberg, "Repetitive DNA and next-generation sequencing: computational challenges and solutions," in *Nature Reviews Genetics*, 2012.

187. A. Udipi *et al.*, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *ISCA*, 2010.

188. A. Udipi *et al.*, "Combining memory and a controller with photonics through 3D-stacking to enable scalable and energy-efficient systems," in *ISCA*, 2011.

189. V. Vasudevan *et al.*, "Using vector interfaces to deliver millions of IOPS from a networked key-value storage server," in *SoCC*, 2012.

190. R. K. Venkatesan *et al.*, "Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *HPCA*, 2006.

191. X. Wang and J. Martinez, "XChange: Scalable dynamic multi-resource allocation in multicore architectures," in *HPCA*, 2015.

192. H.-S. P. Wong *et al.*, "Phase change memory," in *Proceedings of the IEEE*, 2010.

193. H.-S. P. Wong *et al.*, "Metal-oxide RRAM," in *Proceedings of the IEEE*, 2012.

194. M. Xie *et al.*, "Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning," in *HPCA*, 2014.

195. Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *ISCA*, 2009.

196. H. Xin *et al.*, "Accelerating read mapping with FastHASH," in *BMC Genomics*, 2013.

197. H. Xin *et al.*, "Shifted Hamming Distance: A Fast and Accurate SIMD-friendly Filter to Accelerate Alignment Verification in Read Mapping," in *Bioinformatics*, 2015.

198. J. Yang *et al.*, "Frequent value compression in data caches," in *MICRO*, 2000.

199. D. Yoon *et al.*, "Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput," in *ISCA*, 2011.

200. D. Yoon *et al.*, "The dynamic granularity memory system," in *ISCA*, 2012.

201. H. Yoon *et al.*, "Row buffer locality aware caching policies for hybrid memories," in *ICCD*, 2012.

202. H. Yoon *et al.*, "Data mapping and buffering in multi-level cell memory for higher performance and energy efficiency." *CMU SAFARI Tech. Report*, 2013.

203. H. Yoon *et al.*, "Efficient data mapping and buffering techniques for multi-level cell phase-change memories," *TACO*, 2014.

204. J. Zhao *et al.*, "FIRM: Fair and high-performance memory control for persistent memory systems," in *MICRO*, 2014.

205. H. Zhou and T. M. Conte, "Enhancing memory level parallelism via recovery-free value prediction," in *ICS*, 2003.

206. S. Zhuravlev *et al.*, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, 2010.

DOI: 10.14529/jsfi140303

Early evaluation of direct large-scale InfiniBand networks with adaptive routing

***Alexander N. Daryin*[1 2], *Anton A. Korzh*[3]**

We assess the problem of choosing optimal direct topology for InfiniBand networks in terms of performance. Newest topologies like Dragonfly, Flattened butterfly and Slim Fly are considered, as well as standard Tori and Hypercubes. We consider some reasonable extensions to InfiniBand hardware which could be implemented by vendors easily and may allow reasonable routing algorithms for such topologies. A number of routing algorithms are proposed and compared for various traffic patterns. Mapping algorithms for Dragonfly and Flattened Butterfly are proposed. Based on this research it has been decided to use Flattened Butterfly topology for system #22 in November 2014 Top 500 list.

Keywords: adaptive routing, InfiniBand, high-radix topology, network simulation.

Introduction

InfiniBand is a *de facto* industry standard in supercomputing with almost 45% entries on the latest Top 500 list [1]. However, the largest systems on the list tend to have custom interconnection networks, with only 2 of Top 10 systems built using InfiniBand. A possible explanation of this fact is that large number of nodes drives the need for modern cost-effective topologies such as Dragonfly and Flattened Butterfly that are not readily supported in current InfiniBand infrastructure. Moreover, these topologies in turn require advanced routing algorithms, including adaptive routing, that cannot be implemented within InfiniBand specification.

The purpose of this paper is to investigate possible performance of large InfiniBand networks given that some extra features are added to the switches.

1. Topologies

We compare topologies and routing algorithm on the following reference configuration. Network contains $n = 2{,}048$ switches with concentration $C = 8$, thus 16,384 nodes. Each switch has $d = 36$ ports, 8 of those taken by nodes and 28 are available for inter-switch connections. Hardware is assembled in 32 twin racks of 64 switches (512 nodes) each. Hence it is desirable to exploit local connections in groups of 16, 32, or 64 switches to reduce cabling cost. Only direct topologies are considered. Number of switches is fixed to maintain roughly the same cost for different topologies.

Theoretical upper bound [16] on the relative bisection bandwidth β is $\frac{d-2}{2C} + o(1) \approx 162{,}5\%$. Practically constructible graphs with greatest known bisection are Ramanujan graphs [22] with lower bound on bisection $\frac{d-2\sqrt{d-1}}{2C}$ for $d = q + 1$ where q is a prime power. For $q = 25$ this yields a 100% relative bisection. However, such graphs are available in a very few sizes and have intractable structure for designing routing algorithm on them. Therefore we assume that a 50% relative bisection bandwidth obtained on several topologies is a reasonable value.

[1]T-Platforms, Moscow, Russia
[2]National Research University – Higher School of Economics, Moscow, Russia
[3]Micron Technology, Inc., Boise, USA

1.1. Tori

A *torus* is a Cartesian product of cycles. Up to 4D tori may be supported in InfiniBand. For higher dimensions it is impossible to provide deadlock freedom of minimal routing. Torus-2QoS, a routing engine for tori in the Open Subnet Manager (OpenSM), only supports 2D and 3D tori.

We shall use the following notation both for tori and for FlatFly later: N is the number of dimensions; K_n is the size of dimension n; L_n is the width of links within dimension n.

The relative bisection of a torus is then $\beta = \min\{4L_n/CK_n\}$ over $n = \overline{1, N}$, its degree is $d = C + 2\sum_n L_n$. A combinatorial search yields the following two options with the greatest possible bisection:

Dimension	Size	Link Widths	Degree	Diameter	Bisection
3D	$8 \times 16 \times 16$	3, 5, 5	34	20	15,6%
4D	$4 \times 8 \times 8 \times 8$	2, 4, 4, 4	36	14	25,0%

Cayley graphs of commutative groups are a well known alternative of tori, having a similar cabling structure and routing algorithms, but much smaller diameter and greater bisection bandwidth. We do not consider such topologies explicitly, but e.g. a FlatFly is a Cayley graph of a product of cyclic groups with appropriate generators; Dragonfly is a vertex-transitive graph and thus has a structure similar to that of a Cayley graph of a non-commutative group.

1.2. FlatFly

Consider a Cartesian product of N full graphs. Historically, several names were used for this topology:

- *Generalized Hypercube* [5] (this term implies single inter-switch links and one adapter per switch);
- *Flattened Butterfly* [14] (implying single inter-switch links, and equals orders of full graphs equal to concentration);
- *HyperX* [3] (subsumes both previous cases).

We shall refer to this topology as *FlatFly* (FF) and use the same notation as for tori: K_n for orders of full graphs and L_n for link width in each of them, $n = \overline{1, N}$. According to [3], the relative bisection bandwidth of the FlatFly network is $\beta = \min\{K_n L_n\}/2C$. Diameter of a FlatFly is equal to its dimension N, and degree is equal to $C + \sum_n L_n(K_n - 1)$.

Parameters of optimal FlatFly network are determined by a combinatorial search procedure outlined in [3]. For our setup this is a 4D FlatFly with dimensions $4 \times 8 \times 8 \times 8$ and link widths 2, 1, 1, 1. It's diameter is 4, radix is 35, and relative bisection is 50%.

Cabling of this network is relatively easy. A 8×8 part (3rd and 4th dimensions) occupies a twin rack. Then each rack is connected to 3 racks in the first dimension with bundles of 128 cables, and to 7 racks in the second dimension with bundles of 64 cables.

FlatFly may serve as a building block for other topologies. For example, groups in Dragonfly topology are usually connected as a full graph (which is a 1D FlatFly), or a 2D FlatFly [9].

1.3. Hypercube

A *hypercube* is a Cartesian product of 2-vertex complete graphs, and may be considered a particular case of both torus and FlatFly. Unlike tori, dimension of hypercube built with

InfiniBand is only limited by the number of nodes in subnet (which in practice means a maximum of 14D). However, as a FlatFly it usually is not optimal in terms of bisection bandwidth and diameter.

In our setup, hypercube is 11D with double links in each dimension. This gives diameter 11, relative bisection 25%, and radix 30. The remaining 6 ports may be utilized to increase bisection bandwidth within one rack to 37,5%.

1.4. Dragonfly

A *Dragonfly* [15] consists of groups connected as full graphs (with "local" links). These groups, regarded as vertices, are again connected as a full graph (with "global" links).

Denote by K the size of a group, and by G the number of global links per switch. Then there are KG groups containing K^2G switches[4]. A typical shortest path crosses one "global" and two "local" full graphs, hence the relative bisection is $\beta = \min\{K/4C, G/2C\}$. Graph diameter is 3, and degree is $C + G + K - 1$.

It is reasonable to choose K and G such that $K/4C = G/2C$, which gives $K = 16$ and $G = 8$ for our setup. The radix is 30. Unlike other considered topologies, cabling of global links in a Dragonfly is really messy.

1.5. Slim Fly

A *Slim Fly* [4] topology is built with McKay–Miller–Širáň (MMS) graphs [20] parameterized by a prime power q. Its $2q^2$ vertices are elements of linear space $\mathbb{Z}_2 \times \mathbb{F}_q \times \mathbb{F}_q$, where $\mathbb{F}_q$ is a field of order q. Coordinates of vertices are denoted as (t, x, y). There are two kinds of edges:

1. y-edges: (t, x, y_0) is connected with (t, x, y_1) iff $y_1 - y_0 \in X_t$. For $q = 2^p$, the sets X_t are $X_0 = \{1, \xi^2, \xi^4, \dots, \xi^{q-2}\}$ and $X_1 = \{\xi, \xi^3, \dots, \xi^{q-1}\}$, where ξ is a primitive element of $\mathbb{F}_q$.
2. t-edges: $(0, x_0, y_0)$ is connected with $(1, x_1, y_1)$ iff $y_0 = x_0x_1 + y_1$.

Diameter of a MMS graph is 2. For $q = 2^p$ its radix is $C + 3Lq/2$, and we hypothesize that its relative bisection is $\beta = Lq/2C$. Here L is link width.

Unfortunately, an MMS graph with 2,048 switches corresponds to $q = 32$ and requires switches of radix 56. Instead of that, we will consider Cartesian products of Slim Fly and FlatFly. There are two options for our setup:

	Slim Fly q	L	FlatFly K_n	L_n	Degree	Diameter	Bisection
SF×FF-1	4	2	8×8	1, 1	34	4	50%
SF×FF-2	8	1	16	1	35	3	50%

SF×FF-1 replaces 4×8 dimensions of our FlatFly setup with a Slim Fly. This further simplifies cabling: now each twin rack is connected to 6 other racks using bundles of 128 cables. Also, it saves 1,024 optical cables.

SF×FF-2 has the same radix as FlatFly, but a lower diameter of 3, and groups of 16 switches have relative bisection 100%.

[4]Actually there are $KG + 1$ groups and $(KG + 1)K$ switches, but we allow for a small irregularity in topology omitting one global link per group, for the total number of switches to be a power of 2.

2. Mapping algorithms

Some routing algorithms assume that the structure of a subnet is given in advance, and that all switches have their coordinates assigned. This may not be always the case in practice. For example, in OpenSM a routing engine is given only a list of nodes and links, and has to recover network structure on its own.

Here we present mapping algorithms for FlatFly and Dragonfly.

2.1. Mapping a FlatFly

To map a FlatFly, we construct an equivalence relation ε on the set of its links, using the following rules.

1. Two uni-directional links e_1 and e_2 constituting one bi-directional link are equivalent: $e_1 \varepsilon e_2$.
2. Two parallel links e_1 and e_2 connecting the same two switches are equivalent: $e_1 \varepsilon e_2$.
3. Three links e_1, e_2, e_3 forming a rectangle are equivalent: $e_1 \varepsilon e_2$, $e_2 \varepsilon e_3$.
4. If four links $e_1, \ldots, e_4$ form a rectangle, then the opposite sides of this rectangle are equivalent: $e_1 \varepsilon e_3$, $e_2 \varepsilon e_4$.

The total number of traversed edges here is at most nd^2.

The rules above do not define the entire relation ε; we need to compute their transitive closure. This may be done efficiently using the Union-Find algorithm from [25] with $O(n \ln n)$ operations.

Each equivalence class D_i of ε corresponds to a dimension of FlatFly, $i = \overline{1, N}$. Using ε, we then assign coordinates to switches. Additional N equivalence relations ε_i are constructed using the following rule. If edge e connects vertices v_1 and v_2, and $e \in D_i$, then $v_1 \varepsilon_j v_2$ for all $j \neq i$. After calculating the transitive closure, each equivalence class C_{ij} of ε_i corresponds to a single value of i-th coordinate, $j = \overline{1, K_i}$.

The described algorithm can be applied to incomplete topologies. Experiments on our setup have shown that FlatFly is still mapped correctly if 20% switches and 15% links are missing at random. Hypercube is mapped correctly with 15% missing switches and 10% missing links.

2.2. Mapping a Dragonfly

For Dragonfly mapping, we construct the same equivalence relation ε as for FlatFly. Each equivalence class of ε containing more than one edge represents local links of one Dragonfly group. All other links are global links.

In our setup, incomplete Dragonfly is mapped correctly with 40% switches and 10% links missing at random.

3. Routing

InfiniBand [2] fabric is a packet-switching network. Each network device is assigned a subnet address called *Local Identifier* (LID). Each link has several *Virtual Lanes* (VL) used by the routing engine to provide deadlock freedom and quality of service. The route of each packet from source to destination is determined by two header fields set at the source node: Destination LID (DLID) and Service Level (SL).

LID is a 16-bit integer, but the actual address capacity is 48K since 16K addresses are used for multicast groups. SL takes values from 0 to 15. There may be 1, 2, 4, 8, or 15 VLs per link; a common value for modern hardware is 8 VLs.

There are four mechanisms constituting an InfiniBand routing function:

1. Each switch has a *Linear Forwarding Table* (LFT) that maps DLIDs into output ports.
2. After selecting the output port, switch consults an *SL2VL* Table to select output VL depending on input port, output port and SL.
3. Before sending a packet, the source node requests a *Path Record* from the subnet manager, to find out which SL should be used for these source and destination nodes.
4. *LID Mask Control* (LMC) feature allows assigning more than one LID per node. This results in several paths between the same two nodes that may be used for load balancing or other purposes.

In theory [8], one may consider routing functions in a very general form

$$(\text{Source}, \text{Destination}, \text{Switch}, \text{In Port}, \text{In VC}) \to (\text{Out Port}, \text{Out VC}).$$

InfiniBand factors this dependence into

$$\begin{cases} (\text{Source}, \text{Destination}) \to (\text{DLID}, \text{SL}), \\ (\text{Switch}, \text{DLID}) \to \text{Out Port}, \\ (\text{Switch}, \text{SL}, \text{In Port}, \text{Out Port}) \to \text{Out VL}. \end{cases}$$

In particular, the VL chosen does not directly depend on the destination, which limits the choices in the design of deadlock-free routing algorithms.

3.1. Adaptive routing

InfiniBand specification [2] does not support *Adaptive Routing* (AR). Moreover, a number of measures has been taken to ensure that packets arrive in order. On the other hand, it is always possible to find a traffic pattern on which a particular static routing will achieve only a small portion of the available bisection bandwidth [11]. For example, in our setup if 8 nodes on one switch communicate with 8 nodes on a neighbor switch using any minimal routing, they will only get 12,5% bandwidth, even if relative bisection is 50,%.

A number of strategies have been proposed to implement AR in InfiniBand. *Multipath routing* uses LMC and congestion notification mechanism to select a least-congested path at the source [18]. A possible modification to switch hardware [19] would treat all LIDs assigned to the same node interchangeably and dynamically choose output port from LFT entries corresponding to them. Finally, Mellanox claims support of AR in its switches [21].

Here we assume that switch hardware is modified in such a way that for each destination it is possible to specify a *set* of output ports instead of a single port. This allows implementing a *minimal* adaptive routing. However, although it may perform better on some traffic patterns, many other patterns will not benefit from adaptivity if only one shortest path is available between a pair of nodes (which is exactly the case in our example with 8 collocated nodes talking to nodes on a neighbor switch). Furthermore, a fully adaptive minimal routing will in general have credit loops that cannot be eliminated using standard InfiniBand features.

The key problem in designing non-minimal adaptive routing algorithms for InfiniBand is that no information is accumulated in packet header as a packet traverses the fabric. The only header field not covered by the Invariant CRC is VL. However, input VL does not influence neither output port nor output VL. Thus, it is hard, if possible, for a non-minimal routing to guarantee progress towards destination and to avoid livelocks.

Since VL is the only header field that can change from hop to hop, we assume the following modifications to the switch hardware:

1. Output VL is selected using input VL: either it is simply incremented (so called *VL hopping*), or input VL is used instead of SL in the SL2VL mechanism (so that it becomes VL2VL).
2. There is a separate forwarding table for each input VL[5].

With these modifications, the routing function is now described as

$$\begin{cases} (\text{Source}, \text{Destination}) \to (\text{DLID}, \text{VL}), \\ (\text{Switch}, \text{In VL}, \text{DLID}) \to \{\text{Out Port}_1, \ldots, \text{Out Port}_k\}, \\ (\text{Switch}, \text{In VL}, \text{In Port}, \text{Out Port}) \to \text{Out VL}. \end{cases}$$

To reduce latency, it is beneficial to prefer shorter routes when possible. This is not part of a routing, but of a *selection function* that chooses one port from a set based on congestion information. In our simulation we assume that two priorities are available: high priority for minimal paths and low priority for non-minimal. An exception is deflection routing requiring three levels of priority.

We finally note that using AR also requires significant changes to software (including MPI) to properly handle out-of- order packets. This is beyond the scope of this article.

3.2. Torus routing

A deadlock-free routing for tori usually uses two virtual channels to prevent cycles within a dimension[6]: one for packets crossing a dateline in that dimension, another for all other packets. To avoid cycles between dimensions, they are traversed in a fixed order, hence the name *Direction Order Routing* (DOR).

It turns out that dateline crossing at each dimension should be determined at the source node and stored in SL (because VL cannot be chosen per destination address). Since SL has 4 bits, the maximum dimension of InfiniBand tori is 4D.

Summing up, adaptive routing for tori can be only used to select one of the parallel output links.

3.3. FlatFly routing

DOR routing can be applied to FlatFly. Unlike tori, there is no need to use two VLs since a packet traverses at most one link in each dimension.

Adaptive DOR (ADOR) allows an additional hop in each dimension [9, 26]. This is implemented by the following rules:

1. VL 0 is used when input port belongs to a host or to a different dimension than output port.
2. VL 1 is used if input and output ports belong to the same dimension.

[5]Some routing algorithms require a separate forwarding table for each input VL and each input port.

[6]Limitations of InfiniBand routing preclude using other deadlock-avoidance schemes such as turn-based routing. VL hopping is also not an option since the diameter of torus network is larger than the number of available VLs.

3. Packet with input VL 0 is routed to all switches in current dimension, preferring the shortest route.
4. Packet with input VL 1 is routed in current dimension along the shortest route.

DOR and ADOR do not use a large portion of crossbar (all transitions from higher to lower dimensions). *Mixed DOR* uses the remaining two bits of VL to make possible several orderings of dimensions.

1. At the source node, SL is set of $d \mod 4$, where d is the index of the destination host within the destination switch.
2. Based on SL, one of the four dimension orders are used[7]: $(1, 2, 3, 4)$, $(2, 4, 1, 3)$, $(3, 1, 4, 2)$, $(4, 3, 2, 1)$.

Mixed ADOR combines Mixed DOR and ADOR and chooses VL as either $2SL$ (for hops in dimension order) or $2SL + 1$ (for extra hops in the same dimension).

Another pitfall of (A)DOR is that all packets with the same source and destination switches will meet at the intermediate switches. *Twisted DOR* solves this problem by making an obligatory non-minimal hop at the beginning. Assume that we are at the switch with coordinates $(x_1, \dots, x_n)$ and d is the index of the destination host.

1. If a packet comes from a host[8], route it to switch with coordinates $(x_1, \dots, d \mod K_n)$ using VL 1 (if $d = x_n \mod K_n$, do nothing).
2. After that, proceed with DOR using VL 0.

Twisted ADOR is a combination of Twisted DOR and ADOR, with the exception that no additional hop is allowed in the last dimension (since it has been already made at the first step).

Finally, *Twisted Mixed DOR* combines Twisted and Mixed DORs, and the same goes for *Twisted Mixed ADOR*.

3.4. Hypercube routing

As hypercube is a particular case of a FlatFly, we can apply the latter's DOR and Mixed DOR routing algorithms. With only two switches per dimension, ADOR is the same as DOR here. Also, "Twisted" routings did not perform well on simulator, so we don't use them either.

There are special AR algorithms for hypercubes, e.g. described in [10]. We have simulate three of them:

1. *Negative First*: route adaptively in all negative dimensions (those where coordinate of source is 1 and that of destination is 0), then route adaptively in all other dimensions.
2. *All but one (ABO) Negative First*: route adaptively in all negative dimensions except dimension 1, then route adaptively in all other dimensions.
3. *All but one (ABO) Positive Last*: route adaptively in all negative dimension and dimension 1, then route adaptively in all other dimensions.

3.5. Dragonfly routing

Although Dragonfly features low diameter and high relative bisection, advanced routing techniques are required to achieve appropriate network throughput. Indeed, any two groups are connected with only one link, and if all hosts in the first group happen to communicate with all

[7]These permutations where found using integer linear programming minimizing the maximum load on each part of the crossbar.

[8]This algorithm requires selection of forwarding table depending on input port .

hosts in the second one sharing the same link, the resulting throughput will be $1/KC$, which is less than 1% for our setup.

A popular Dragonfly routing UGAL is based on a scheme proposed by Valiant [27]: most packets are first routed to a randomly chosen intermediate group, and only then to the destination. However, there is no hardware support for Valiant-type routings in InfiniBand, so it is impossible to implement UGAL together with RDMA.

The basic *Static* routing for Dragonfly is[9]:

1. in source group, make a local hop to the node that has a link to the destination group;
2. make a global hop;
3. in destination group, make a local hop to the destination switch.

Here we use a family of adaptive routing algorithms that use only local congestion information. They are coded by a combination of letters G, S, I, and D, each allowing extra non-minimal hops of different kinds:

G: extra global hop is allowed (leading to an intermediate group);
S: extra local hop is allowed in the source group;
I: extra local hop is allowed in the intermediate group;
D: extra local hop is allowed in the destination group.

At most 8 total hops are possible. Up to 6 VLs are required: VLs 0 and 1 are used in the source group, 2 and 3 in the intermediate group, 4 and 5 in the destination group. Higher priority is assigned to minimal routes.

3.6. SlimFly and SF×FF routing

Here we describe the structure of the minimal routing in Slim Fly. Consider the following cases when routing a packet from (t_s, x_s, y_s) to (t_d, x_d, y_d).

1. $t_s = t_d = t$, $x_s = x_d = x$. Then either $y_s - y_d \in X_t$, or there exists $y_i \in F_q$ such that $y_s - y_i \in X_t$ and $y_i - y_d \in X_t$ (by construction of X_t). Depending on that, the shortest route is either "y" or "yy" (encoded by the link types). Note that it cannot be one of "ty" or "yt" since $t_s = t_d$. Also it cannot be "tt" as will be seen below.
2. $t_s = t_d = t$, $x_s \neq x_d$. Since y-links do not change x, the only possible option is the "tt" route. Denote by (t_i, x_i, y_i) the intermediate vertex, $t_i = 1 - t$. Then[10] if $t = 0$,

$$\begin{cases} y_s = x_s x_i + y_i; \\ y_d = x_d x_i + y_i \end{cases} \implies \begin{cases} x_i = (y_s - y_d)/(x_s - x_d), \\ y_i = y_s - x_s x_i. \end{cases} \tag{1}$$

If $t = 1$,

$$\begin{cases} y_i = x_s x_i + y_s; \\ y_i = x_d x_i + y_d \end{cases} \implies \begin{cases} x_i = -(y_s - y_d)/(x_s - x_d), \\ y_i = y_s + x_s x_i. \end{cases} \tag{2}$$

Note that these systems are not solvable if $x_s = x_d$. All arithmetic is in the field F_q. Relations (1) and (2) may be unified as

$$\begin{cases} x_i = (-1)^t (y_s - y_d)/(x_s - x_d), \\ y_i = y_s + (-1)^t x_s x_i. \end{cases}$$

[9]Note that this routing is not minimal, as some shortest paths consist of two global hops.
[10]All arithmetic here is in $\mathbb{F}_q$.

3. $t_s \neq t_d$. If $(t_d - t_s)(y_s - y_d) = x_s x_d$, then the shortest route is the corresponding t-link. Otherwise it should be either "*ty*" or "*yt*".
   (a) "*ty*". The intermediate node is (t_d, x_d, y_i), and then
   $$y_s - y_i = (t_d - t_s)x_s x_d \quad \Longrightarrow \quad y_i = y_s - (t_d - t_s)x_s x_d.$$
   given that $y_i - y_d \in X_{t_d}$.
   (b) "*yt*". The intermediate node is (t_s, x_s, y_i), and then
   $$y_i - y_d = (t_d - t_s)x_s x_d \quad \Longrightarrow \quad y_i = y_d + (t_d - t_s)x_s x_d.$$
   given that $y_s - y_i \in X_{t_s}$.
   At least one of two cases should hold since the diameter of MMS graph is 2.

As seen from the minimal routing analysis, the number of shortest paths between (t_s, x_s, y_s) and (t_d, x_d, y_d) is

- 1 or 2 if $t_s \neq t_d$ (but the case of 1 is encountered more often);
- exactly one of $t_s = t_s$, $x_s \neq x_d$;
- usually multiple if $t_s = t_d$, $x_s = x_d$.

Experiments show that most source-destination pairs only have 1 shortest path between them.

To avoid deadlocks, on two-hop paths VL 0 is used on the first hop and VL 1 on the second. This may be implemented using standard SL2VL feature: if a packet comes from a host, select VL 0, otherwise select VL 1.

DOR, ADOR and their Mixed and Twisted variants may be used for SF×FF products, considering Slim Fly as a dimension of a Flat Fly.

3.7. Topology agnostic algorithms

There exists a number of static generic routing algorithms for InfiniBand: UpDn [17], DF-SSSP [7], LASH [23]. Achieving deadlock freedom for an arbitrary topology is challenging, and mentioned algorithms do not provide such a guarantee.

Under our assumptions, VL hopping may be used to provide deadlock freedom. Indeed, no credit loops are possible if VLs are used in strictly increasing order. This works when maximum path length does not exceed the number of available VLs. In our setup, we can use VL hopping for FlatFly, Dragonfly and SF×FF.

We use a family of algorithms that we call *Distance Based Routing.* Its parameters are E – the number of extra hops allowed, and F – the flag indicating whether deflection is permitted.

Suppose that shortest distance between source and destination switches is $D > 0$, and the remaining distance at current switch is $d > 0$.

1. Source host assigns $\mathrm{VL} = D + E \mod 8$ to the packet.
2. On each hop, VL is decremented $\mod 8$.
3. For each output port p leading to a switch, calculate distance from that switch to the destination switch, d_p.
   (a) If $d > d_p$, route to p with high priority (shortest path).
   (b) If $d = d_p \leq \mathrm{VL}_{\mathrm{out}}$, route to p with medium priority (routing sideways).
   (c) If $d < d_p \leq \mathrm{VL}_{\mathrm{out}}$ and F is set, route to p with low priority (deflection).

In these terms, fully adaptive minimal routing corresponds to $E = 0$ and is called *Distance Based.* Non-minimal routing ($E > 0$) without deflection is denoted as *Distance Based + E.*

Finally, routing with deflection is denoted *Deflection + E*. In the latter case $E \geq 2$, otherwise enabling deflection makes no sense.

4. Simulation results

In this section, we compare performance of described topologies and routing algorithms. For large-scale networks, we have developed a parallel event-driven simulator following methodology described in [6]. It has been verified against BookSim and real-world data and produces consistent results.

The following table contains relative saturation bandwidth [6] for each topology, routing algorithm and traffic pattern. Since network size is a power of two, we use bit patterns for comparison: complement, reverse, rotation, shuffle, and transpose. There is also uniform (random all-to-all) pattern.

				Bit		
Routing	Uniform	Cmpl	Rvrs	Rotn	Shuf	Trns
Torus						
DOR 3D	26,5%	15,6%	7,6%	7,0%	7,6%	7,1%
DOR 4D	48,2%	25,0%	2,9%	17,0%	12,2%	2,9%
FlatFly						
DOR	81,3%	11,7%	1,6%	12,5%	7,8%	2,3%
Mixed DOR	89,8%	11,7%	5,5%	7,8%	5,5%	2,3%
Twisted DOR	24,2%	11,7%	11,7%	11,7%	12,5%	5,5%
Twisted Mixed DOR	46,9%	12,5%	11,7%	9,0%	10,9%	2,3%
ADOR	52,3%	50,0%	3,1%	25,0%	24,2%	2,3%
Mixed ADOR	56,3%	50,0%	10,2%	24,2%	24,2%	5,5%
Twisted ADOR	24,2%	50,0%	21,1%	24,2%	24,2%	11,7%
Twisted Mixed ADOR	53,1%	28,1%	22,7%	24,2%	24,2%	5,5%
Distance Based	93,8%	11,7%	9,4%	11,7%	9,4%	10,2%
Distance Based +1	95,3%	11,7%	14,8%	24,2%	24,2%	24,2%
Distance Based +2	94,5%	24,2%	14,8%	24,2%	24,2%	24,2%
Distance Based +3	95,3%	24,2%	15,6%	39,8%	24,2%	24,2%
Distance Based +4	96,1%	50,8%	14,8%	39,8%	37,5%	37,5%
Deflection +2	96,1%	24,2%	24,2%	24,2%	24,2%	24,2%
Deflection +3	94,5%	24,2%	24,2%	50,0%	24,2%	24,2%
Deflection +4	94,5%	50,8%	24,2%	50,0%	24,2%	24,2%
Hypercube						
DOR	24,2%	24,2%	0,8%	24,2%	11,7%	0,8%
Mixed DOR	24,2%	24,2%	2,3%	11,7%	11,7%	0,8%
Negative First	2,3%	0,0%	0,8%	0,8%	0,8%	0,8%
ABO Negative First	5,5%	0,0%	0,8%	0,8%	0,8%	0,8%
ABO Positive Last	5,5%	0,0%	0,8%	0,8%	0,8%	0,8%
Dragonfly						
Static	24,2%	5,5%	5,5%	5,5%	5,5%	5,5%
S	64,8%	0,0%	11,7%	0,8%	0,8%	9,4%
D	68,8%	0,0%	11,7%	0,8%	0,8%	10,9%

Routing	Uniform	Cmpl	Rvrs	Bit Rotn	Shuf	Trns
SD	50,8%	0,0%	11,7%	0,8%	0,8%	11,7%
G	11,7%	5,5%	5,5%	5,5%	5,5%	5,5%
GS	11,7%	5,5%	11,7%	5,5%	5,5%	9,4%
GI	11,7%	11,7%	5,5%	10,9%	11,7%	5,5%
GD	11,7%	11,7%	5,5%	11,7%	21,9%	5,5%
GSI	22,7%	11,7%	5,5%	10,9%	11,7%	5,5%
GID	11,7%	46,9%	5,5%	24,2%	24,2%	5,5%
GSD	11,7%	11,7%	11,7%	11,7%	11,7%	10,9%
GSID	23,4%	24,2%	5,5%	24,2%	24,2%	5,5%
Distance Based	82,0%	0,0%	5,5%	0,8%	0,8%	5,5%
Distance Based +1	61,7%	0,8%	10,9%	0,8%	2,3%	11,7%
Distance Based +2	59,4%	0,8%	11,7%	2,3%	2,3%	9,4%
Distance Based +3	50,0%	0,8%	11,7%	2,3%	2,3%	10,9%
Distance Based +4	24,2%	2,3%	11,7%	2,3%	2,3%	10,9%
Deflection +2	57,0%	0,8%	5,5%	2,3%	2,3%	9,4%
Deflection +3	48,4%	2,3%	11,7%	2,3%	5,5%	11,7%
Deflection +4	24,2%	2,3%	11,7%	5,5%	5,5%	11,7%
SF×FF-1						
Distance Based	80,5%	11,7%	11,7%	10,9%	5,5%	9,4%
Distance Based +1	80,5%	11,7%	39,1%	11,7%	14,8%	11,7%
Distance Based +2	80,5%	19,5%	30,5%	11,7%	11,7%	11,7%
Distance Based +3	80,5%	19,5%	30,5%	11,7%	11,7%	11,7%
Distance Based +4	80,5%	19,5%	30,5%	11,7%	11,7%	11,7%
Deflection +2	82,0%	11,7%	24,2%	23,4%	21,9%	24,2%
Deflection +3	81,2%	19,5%	24,2%	24,2%	24,2%	32,8%
Deflection +4	80,5%	21,9%	24,2%	24,2%	24,2%	32,8%
SF×FF-2						
Distance Based	61,7%	5,5%	5,5%	5,5%	5,5%	5,5%
Distance Based +1	61,7%	11,7%	14,8%	7,0%	7,0%	7,0%
Distance Based +2	61,7%	11,7%	14,8%	7,0%	7,0%	8,6%
Distance Based +3	61,7%	11,7%	17,2%	7,0%	7,0%	8,6%
Distance Based +4	71,9%	11,7%	17,2%	7,0%	7,0%	7,0%
Deflection +2	61,7%	10,2%	11,7%	11,7%	10,9%	11,7%
Deflection +3	61,7%	11,7%	24,2%	11,7%	11,7%	23,4%
Deflection +4	74,2%	11,7%	18,8%	11,7%	10,9%	11,7%

As expected, tori and hypercube perform worse than other topologies on uniform traffic due to lower relative bandwidth.

For FlatFly, the best routing is “Distance Based +4”. Comparing its results with “Distance Based +3” and ADOR family, we conclude that the optimal number of extra hop is equal to dimension of FlatFly, and it’s critical that these hops are made without particular dimension order. Deflection helps to achieve full bisection bandwidth on some patterns, but reduces bandwidth for others.

Hypercube shows low performance on bit reverse and bit transpose patterns. Turn model routings (negative first/ positive last) perform really badly, as they create significant imbalance in packet distribution across the network.

Results for Dragonfly are mixed, with GID and GSD routings showing best results for bit traffic patterns and low performance on uniform traffic at the same time. Distance based and deflection routings perform worse than our specialized algorithms.

Results for SF$\times$FF-1 are better than for SF$\times$FF-2. Deflection helps to break the 12,5% barrier on three traffic patterns, with "Deflection +4" showing the best results for almost all traffic patterns.

Comparing different topologies, FlatFly and SF$\times$FF-1 seem to be the winners, according to the following criteria

1. performance on uniform traffic;
2. performance of best routing on bit traffic patterns;
3. performance of "Distance Based" routing[11];

5. Related work

A performance comparison of various types of interconnects is given in [13]. In [28, 29] authors analyze and simulate adaptive routing in InbiniBand fat trees. Improving performance of large-scale InfiniBand networks through optimized task placement is subject of [24]. DFSSSP routing [7, 12] optimizes InfiniBand fabric performance by statically balancing network paths.

6. Conclusions

In this paper, we have analyzed possible topologies for large-scale InfiniBand systems (including tori, hypercube, Dragonfly, Flattened Butterfly, Slim Fly). In most cases, adaptive routing is required in order to achieve theoretical bandwidth limits. We analyze standard InfiniBand routing and list a minimum set of features that should be added in order to support adaptive routing. We describe specialized adaptive routing algorithms for each topology, and a family of topology-agnostic (distance-based) routings. Then we provide simulation results for considered topologies and routing algorithms. Best performance results are shown by Flattened Butterfly and a combination of Flattened Butterfly and Slim Fly.

This work has been partially supported by the Russian Ministry of Education and Science (project # 14.579.21.0074).

References

1. November 2014 TOP500 list (`http://top500.org/lists/2014/11/`).

2. *InfiniBand^{TM} Architecture Specification*, volume 1. IBTA, 2007.

[11]Results for "Distance Based" routing give an upper bound for any static minimal routing. Similarly, results for DOR and Mixed DOR give an upper bound for static DOR-based algorithms.

3. J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of SC'09*. IEEE, Nov 2009.
4. M. Besta and T. Hoefler. Slim Fly: A cost effective low-diameter network topology. In *Proceedings of SC'14*, pages 348–359. IEEE, Nov 2014.
5. L. N. Bhuyan and D. P. Agrawal. Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on Computers*, 33(4):323–333, Apr 1984.
6. W. J. Dally and B. P. Towles. *Principles and Practices of Interconnection Networks*. Elsevier Science, 2003.
7. J. Domke, T. Hoefler, and W. E. Nagel. Deadlock-free oblivious routing for arbitrary topologies. In *Proceedings of IPDPS-11*, pages 616–627. IEEE, May 2011.
8. J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. Elsevier Science, 2002.
9. G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray Cascade: a scalable HPC system based on a Dragonfly network. In *Proceedings of SC'12*, pages 1–9. IEEE, Nov 2012.
10. C. J. Glass and L. M. Ni. The turn model for adaptive routing. *Journal of the ACM*, 41(5):874–902, Sep 1994.
11. T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Proceedings of CLUSTER 2008*, pages 116–125. IEEE, Sep 2008.
12. T. Hoefler, T. Schneider, and A. Lumsdaine. Optimized routing for large-scale infiniband networks. In *Proceedings of HOTI'09*, pages 103–111, Aug 2009.
13. D. J. Kerbyson, K. J. Barker, A. Vishnu, and A. Hoisie. A performance comparison of current HPC systems: Blue Gene/Q, Cray XE6 and InfiniBand systems. *Future Generation Computer Systems*, 30:291–304, Jan 2014.
14. J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of ISCA'07*, number 13, pages 126–137. ACM, May 2007.
15. J. Kim, W. J. Dally, S. L. Scott, and D. Abts. Cost-efficient Dragonfly topology for large-scale systems. *IEEE Micro*, 29(1):33–40, 2008.
16. A. V. Kostochka and L. S. Melnikov. On bounds of the bisection width of cubic graphs. In J. Nesetril and M. Fiedler, editors, *Proceedings of the Fourth Czechoslovakian Symposium on Combinatorics, Graphs and Complexity*, volume 51 of *Annals of Discrete Mathematics*, pages 151–154. Elsevier, 1992.
17. P. Lopez, J. Flich, and J. Duato. Deadlock-free routing in InfiniBand$^{\text{TM}}$ through destination renaming. In *Proceedings of ICPP-01*, pages 427–434. IEEE, Sep 2001.
18. D. F. Lugones, D. Franco, and E. Luque. Dynamic routing balancing on infiniband network. *Journal of Computer Science & Technology*, 8(2):104–110, Jul 2008.

19. J. C. Martinez, J. Flich, A. Robles, P. Lopez, and J. Duato. Supporting fully adaptive routing in InfiniBand networks. In *Proceedings of IPDPS-03*. IEEE, Apr 2003.

20. B. D. McKay, M. Miller, and J. Širáň. A note on large graphs of diameter two and given maximum degree. *Journal of Combinatorial Theory, Series B*, 74(1):110–118, Sep 1998.

21. Mellanox Technologies. SwitchX product brief (`http://www.mellanox.com/related-docs/prod_silicon/SwitchX_VPI.pdf`).

22. M. Morgenstern. Existence and explicit constructions of $q + 1$ regular Ramanujan graphs for every prime power q. *Journal of Combinatorial Theory, Series B*, 62(1):44–62, Sep 1994.

23. T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (LASH) routing in irregular system area networks. In *Proceedings of IPDPS-02*, pages 162–169, Apr 2002.

24. H. Subramoni, S. Potluri, K. C. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda. Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes. In *Proceedings of SC'12*. IEEE, Nov 2012.

25. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, Apr 1975.

26. A. Thamarakuzhi and J. A. Chandy. 2-dilated flattened butterfly: A nonblocking switching topology for high-radix networks. *Computer Communications*, 34(15):1822–1835, Sep 2011.

27. L. G. Valiant. A scheme for fast parallel communication. *SIAM journal on computing*, 11(2):350–361, May 1982.

28. E. Zahavi, I. Keslassy, and A. Kolodny. Distributed adaptive routing for big-data applications running on data center networks. In *Proceedings of ANCS'12*, pages 99–110. ACM, 2012.

29. E. Zahavi, I. Keslassy, and A. Kolodny. *Distributed Adaptive Routing Convergence to Non-Blocking DCN Routing Assignments*, volume 32, pages 88–101. Jan 2014.

DOI: 10.14529/jsfi140304

Heterogeneous Parallel Computing: from Clusters of Workstations to Hierarchical Hybrid Platforms

A.L. Lastovetsky[1]

The paper overviews the state of the art in design and implementation of data parallel scientific applications on heterogeneous platforms. It covers both traditional approaches originally designed for clusters of heterogeneous workstations and the most recent methods developed in the context of modern multicore and multi-accelerator heterogeneous platforms.

Keywords: parallel computing, heterogeneous computing, data partitioning.

Introduction

High performance computing systems become increasingly heterogeneous and hierarchical. A typical compute node integrates multiple (possibly heterogeneous) cores as well as hardware accelerators such as Graphics Processing Units. The integration is often hierarchical. The motivation behind such complicated architecture is to make these systems more energy efficient. The energy consideration is paramount as future large-scale cluster infrastructures will have to have hundreds of thousands of compute nodes to solve Exascale problems and would not be energy sustainable if nodes of traditional architecture were used. Future large-scale systems will exhibit multiple forms of architectural and non-architectural heterogeneity as well as mean-time-to-failure of minutes. How to develop parallel applications and software that efficiently utilize highly heterogeneous and hierarchical computing and communication resources, while scaling them towards Exascale, maintaining a sustainable energy footprint, and preserving correctness are highly challenging and open questions.

Heterogeneous parallel computing is the area that emerged in 1990s to address the challenges posed by ever increasing heterogeneity and complexity of the HPC platforms. This paper overviews the development of heterogeneous parallel computing technologies as they followed the evolution of heterogeneous HPC platforms from simple single-switched heterogeneous clusters of (uniprocessor) workstations to modern hierarchical clusters of heterogeneous hybrid nodes. It mainly focuses on the design of fundamental data partitioning algorithms supporting the development of data parallel applications able to automatically tune to the executing heterogeneous platform achieving optimal performance (and energy) efficiency. Data parallel applications are the main target of parallel computing technologies because they dominate the scientific and engineering computing domain, as well as the emerging domain of large-scale ("Big") data analytics.

Optimization of data parallel applications on heterogeneous platforms is typically achieved by balancing the load of the heterogeneous processors and minimizing the cost of moving data between the processors. Data partitioning algorithms solve this problem by finding the optimal distribution of data between the processors. They typically require a priori information about the parallel application and platform. Data partitioning is not the only technique used for load balancing. Dynamic load balancing, such as task queue scheduling and work stealing [5, 9, 26, 39–41] balance the load by moving fine-grained tasks between processors during the calculation. Dynamic algorithms do not require a priori information about execution but may incur significant

[1]University College Dublin, Dublin, Ireland

communication overhead on distributed-memory platforms due to data migration. At the same time, dynamic algorithms often use static data partitioning for their initial step to minimize the amount of data redistributions needed. For example, in the state-of-the-art load balancing techniques for multi-node, multicore, and multi-GPU platforms, the performance gain is mainly due to better initial data partitioning. It was shown that even the static distribution based on simplistic performance models (single values specifying the maximum performance of a dominant computational kernel on CPUs and GPUs) improves the performance of traditional dynamic scheduling techniques by up to 250% [44]. In this overview we focus on parallel scientific applications, where computational workload is directly proportional to the size of data, and dedicated HPC platforms, where: (i) the performance of the application is stable in time and is not affected by varying system load; (ii) there is a significant overhead associated with data migration between computing devices; (iii) optimized architecture-specific libraries implementing the same kernels may be available for different computing devices. On these platforms, for most scientific applications, static load balancing algorithms outperform dynamic ones because they do not involve data migration. Therefore, for the type of applications and platforms we focus on, data partitioning is the most appropriate optimization technique.

One very important aspect of optimization of parallel applications on distributed-memory heterogeneous platforms – optimization of their communication cost, is not covered in this paper. A recent analytical overview of methods for optimization of collective communication operations in heterogeneous networks can be found in [21].

1. Optimization of parallel applications on heterogeneous clusters of workstations

1.1. Data partitioning algorithms based on constant performance models

Since the late 1990s, when the first pioneering works in the field were published, the design of heterogeneous parallel algorithms has made a significant progress. At that time, the main target platform for the heterogeneous parallel algorithms being developed was a heterogeneous cluster of workstations, and the simplest possible performance model of this platform was used in the algorithm design. Namely, it was seen as a set of independent heterogeneous (uni)processors, each characterized by a single positive number representing its speed. The speed of the processors can be absolute or relative. The absolute speed of the processors is understood as the number of computational units performed by the processor per one time unit. The relative speed of the processor can be obtained by the normalization of its absolute speed. While this performance model has no communication-related parameters, it still allows for optimization of the communication cost through the minimization of the amount of data moved between processors. This model is also known as Constant Performance Model, or CPM.

Using the CPM, a fundamental problem of optimal distribution of independent equal units of computation over a set of heterogeneous processors was formulated and solved in [7]. The algorithm [7] solving this problem is of complexity $O(p^2)$ and only needs relative speeds. This algorithm is a basic building block in many heterogeneous parallel and distributed algorithms.

This is typical in the design of heterogeneous parallel algorithms that the problem of distribution of computations in proportion to the speed of processors is reduced to the problem of partitioning of some mathematical objects, such as sets, matrices, graphs, etc. Most of the CPM-based algorithms designed so far have been aimed at numerical linear algebra. For example, the problem of LU factorization of a dense matrix A was reduced to the problem of optimal mapping of its column panels $a_1, \ldots, a_n$ to p heterogeneous processors, and the latter problem was further reduced to the problem of partitioning of a well-ordered set (whose elements represent the column panels). Two efficient algorithms solving this partitioning problem have been proposed — the Dynamic Programming (DP) algorithm [7, 10] and the Reverse algorithm [34]. The latter is more suitable for extension to more complex heterogeneous performance models. Other algorithms of partitioning of well-ordered sets, e.g. [6], do not guarantee the return of an optimal solution.

As matrices are probably the most widely used mathematical objects in scientific computing, most of data-partitioning studies deal with them. Matrix partitioning problems occur during the design of parallel linear algebra algorithms for heterogeneous platforms. A typical heterogeneous linear-algebra algorithm is designed as a modification of its homogeneous prototype, and its design is eventually reduced to the problem of optimally partitioning a matrix over heterogeneous processors. From the partitioning point of view, a dense matrix is an integer-valued rectangular. Therefore, if we are only interested in an asymptotically optimal solution (which is typically the case), the problem of its partitioning can be reduced to a problem of the partitioning of a real-valued rectangle.

In a general form, the related geometrical problem has been formulated as follows [8]: Given a set of p processors P_1, P_2, ..., P_p, the relative speed of each of which is characterized by a positive constant, s_i, partition a unit square into p rectangles so that:

- there is a one-to-one mapping between the rectangles and the processors;
- the area of the rectangle allocated to processor P_i is equal to s_i;
- the partitioning minimizes the sum of half-perimeters of the rectangles.

This formulation is motivated by the SUMMA matrix multiplication algorithm [23] and aimed at balancing the load of the processors and minimization of the total volume of data communicated between the processors. Fig. 1 shows one iteration of the heterogeneous SUMMA algorithm assuming that matrices A, B and C are identically partitioned into rectangular submatrices. At each iteration of the main loop, pivot block column of matrix A and pivot block row of matrix B are broadcast horizontally and vertically, then all processors update their own parts of matrix C in parallel. The blocking factor b is a parameter used to adjust the granularity of communications and computations [13], whose optimal value can be found experimentally.

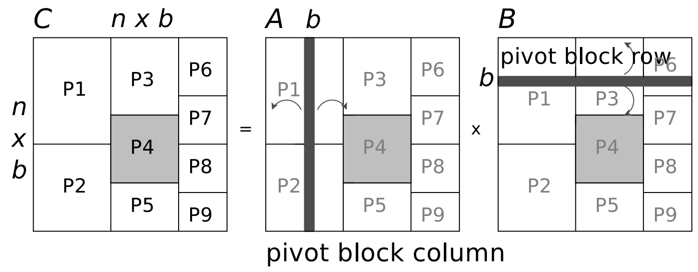


Figure 1. Heterogeneous parallel matrix multiplication

This geometrical partitioning problem is NP-complete [8], but many restricted and practically important versions of this problem have been efficiently solved. The least restrictive is probably the column-based problem looking for an optimal partitioning, the rectangles of which make up columns as illustrated in Fig. 2. An algorithm of the complexity $O(p^3)$ was proposed in [8]. More restricted forms of the column-based geometrical partitioning problem have also been addressed. The pioneering result in the field was a linear algorithm [27] additionally assuming that the number of columns c in the partitioning and the number of rectangles in each column are given. A column-based partitioning with the same number of rectangles in each column is known as a grid-based partitioning. An algorithm of the complexity $O(p^{3/2})$ solving the grid-based partitioning problem was proposed in [29].

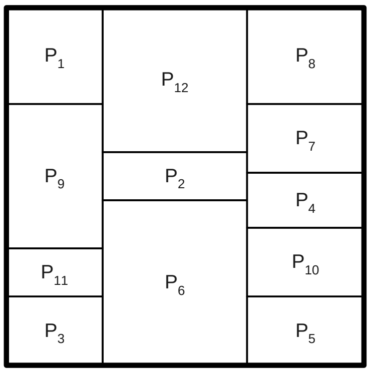


Figure 2. Column-based partitioning of the unit square into 12 rectangles. The rectangles of the partitioning form three columns

A partitioning whose rectangles make both columns and rows is known as a Cartesian partitioning. It is attractive from the implementation point of view because of its very simple and scalable communication pattern. However, the related partitioning problems are very difficult and very little has been achieved in addressing them so far [7].

More recent research [19, 20] challenged the optimality of the rectangular matrix partitioning. Using a specially developed mathematical technique and five different parallel matrix multiplication algorithms, it was proved that the optimal partition shape can be non-rectangular, and the full list of optimal shapes for the cases of two and three processors was identified. Fig. 3 shows these for the case of three processors. The performance model used in this work combined the CPM and the Hockney communication model [24]. These results have a potential to significantly improve the performance of matrix computations on platforms that can be modeled by a small number of interconnected heterogeneous abstract processors, such as hybrid CPU/GPU nodes and clusters of clusters.

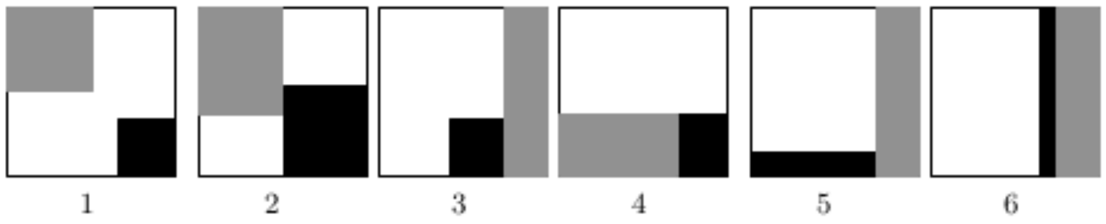


Figure 3. The candidate partition shapes previously identified as potentially optimal three processor shapes. Processors P, R, and S are in white, grey, and black, respectively. (1) Square Corner (2) Rectangle Corner (3) Square Rectangle (4) Block 2D Rectangular (5) L Rectangular (6) Traditional 1D Rectangular

Significant work has been done in partitioning algorithms for graphs, which are then applied to sparse matrices and meshes, the mathematical objects widely used in many scientific applications, e.g. computational fluid dynamics. Algorithms implemented in ParMetis [28], SCOTCH [12], JOSTLE [45] reduce the number of edges between the target subdomains, aiming to minimize the total communication cost of the parallel application. Algorithms implemented in Zoltan [11], PaGrid [4] try to minimize the execution time of the application. All these graph partitioning libraries use performance models combining the CPM and the Hockney model. The models have to be provided by the users.

1.2. Data partitioning algorithms based on functional performance models

The CPM can be a sufficiently accurate approximation of the performance of heterogeneous processors executing a data parallel application if: (i) the processors are general-purpose and execute the same code, (ii) the local tasks are small enough to fit in the main memory but large enough not to fully fit in the processor cache. However, if we consider essentially heterogeneous processors using different code to solve the same task locally, or allow the tasks to span different levels of memory hierarchy on different processors, then the relative speed of the processors can significantly differ for different task sizes. In these situations, the CPM becomes inaccurate, and its use can lead to highly imbalanced load distribution [16]. To address this challenge, a functional performance model (FPM) [35, 37, 38] was proposed. The FPM represents the speed of a processor by a function of problem size. It is built empirically and integrates many important features characterizing the performance of both the architecture and the application. The speed is defined as the number of computation units processed per second. The computation unit can be defined differently for different applications. The important requirement is that its size (in terms of arithmetic operations) should not vary during the execution of the application. One FLOP is a simplest example of computation unit.

The fundamental problem of optimal distribution of n independent equal units of computation between p heterogeneous processors represented by their speed functions was formulated, and very efficient geometrical algorithms (of complexities $O(p^2 \log_2 n)$ and $O(p \log_2 n)$) solving this problem under different assumptions about the shape of the speed functions were proposed [31, 35]. These algorithms are based on the following observation. Let the speed of processor P_i be represented by continuous function $s_i(d) = \frac{d}{t_i(d)}$, where $t_i(d)$ is the execution time for processing of d computation units on the processor P_i. Then the optimal solution of this problem, which balances the load of the processors, will be achieved when all processors execute their work within the same time: $t_1(d_1) = ... = t_p(d_p)$. This can be expressed as:

$$\frac{d_1}{s_1(d_1)} = ... = \frac{d_p}{s_p(d_p)} \text{ , where } d_1 + d_2 + ... + d_p = n \tag{1}$$

The solution to these equations, $d_1, ..., d_p$, can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system as illustrated in Fig. 4

The geometrical algorithms proceed as follows. As any line passing through the origin and intersecting the speed functions represents an optimum distribution for a particular problem size, the space of solutions of the problem (1) consists of all such lines. The two outer bounds of the solution space are selected as the starting point of algorithm. The upper line represents the optimal data distribution $x_1^u, ..., x_p^u$ for some problem size $n_u < n$, $n_u = x_1^u + ... + x_p^u$, while the lower line gives the solution $x_1^l, ..., x_p^l$ for $n_l > n$, $n_l = x_1^l + ... + x_p^l$. The region between two lines is iteratively bisected as shown in Fig. 5.

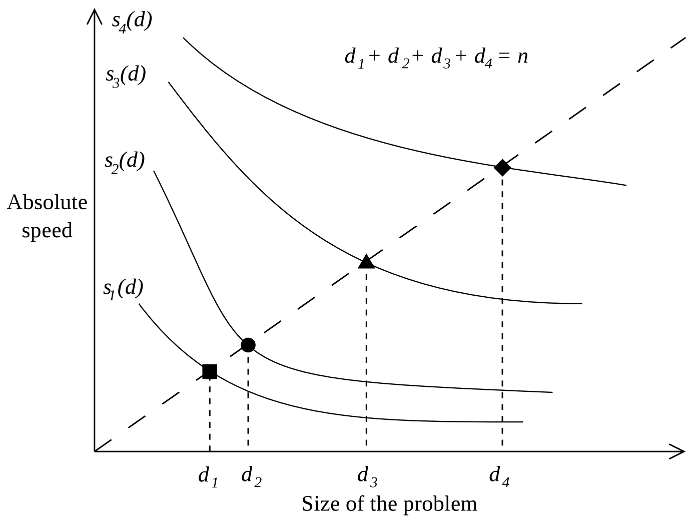


Figure 4. Optimal distribution of computational units showing the geometric proportionality of the number of computation units to the speeds of the processors

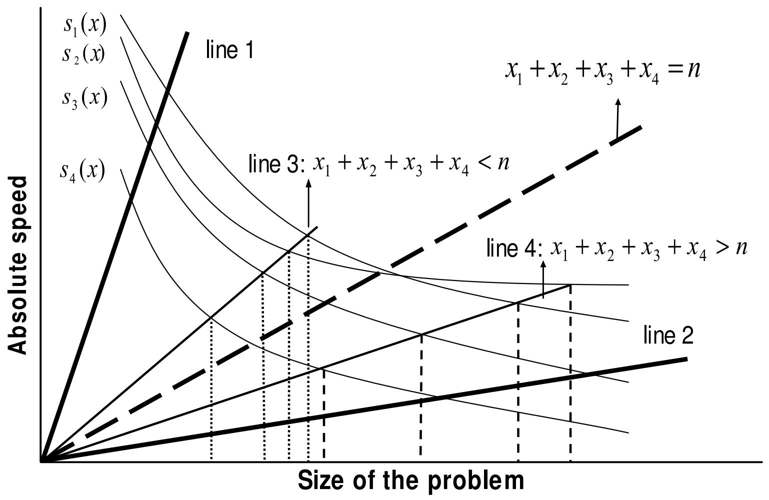

$s_1(x)$
$s_2(x)$
$s_3(x)$
$s_4(x)$
line 1
$x_1 + x_2 + x_3 + x_4 = n$
line 3: $x_1 + x_2 + x_3 + x_4 < n$
line 4: $x_1 + x_2 + x_3 + x_4 > n$
line 2
Absolute speed
Size of the problem

Figure 5. Geometrical data partitioning algorithm. Line 1 (the upper line) and line 2 (the lower line) represent the two initial outer bounds of the solution space. Line 3 represents the first bisection. Line 4 represents the second one. The dashed line represents the optimal solution

At the iteration k, the problem size corresponding to the new line intersecting the speed functions at the points $x_1^k, ..., x_p^k$ is calculated as $n_k = x_1^k + ... + x_p^k$. Depending on whether n_k is less than or greater than n, this line becomes a new upper or lower bound. Making n_k close to n, this algorithm finds the optimal partition of the given problem $x_1, ..., x_p$: $x_1 + ... + x_p = n$. The geometrical algorithms will always find a unique optimal solution if the speed functions satisfy the following assumptions:

1. On the interval $[0, X]$, the function is monotonically increasing and concave.
2. On the interval $[X, \infty]$, the function is monotonically decreasing.

Extensive experiments with many scientific kernels on different workstations have demonstrated that, in general, processor speed can be approximated, within some acceptable degree of accuracy, by a function satisfying these assumptions.

Another algorithm [43] significantly relaxes the restrictions on the shape of speed functions but does not always guarantee the globally optimal solution. This algorithm assumes that the

Akima spline interpolation [1] is used to approximate the speed function. Then it formulates the problem of optimal data partitioning in the form of a system of non-linear equations and applies multidimensional solvers to numerical solution of this system. The algorithm is iterative and always converges in a finite number of iterations returning a solution that balances the load of the processors. The number of iterations depends on the shape of the functions. In practice, the number can be as little as 2 iterations for very smooth speed functions and up to 30 iterations when partitioning in regions of rapidly changing speed functions. For illustration, Fig. 6 shows speed function approximations used in the geometrical algorithms and in the algorithm based on the multidimensional solvers.

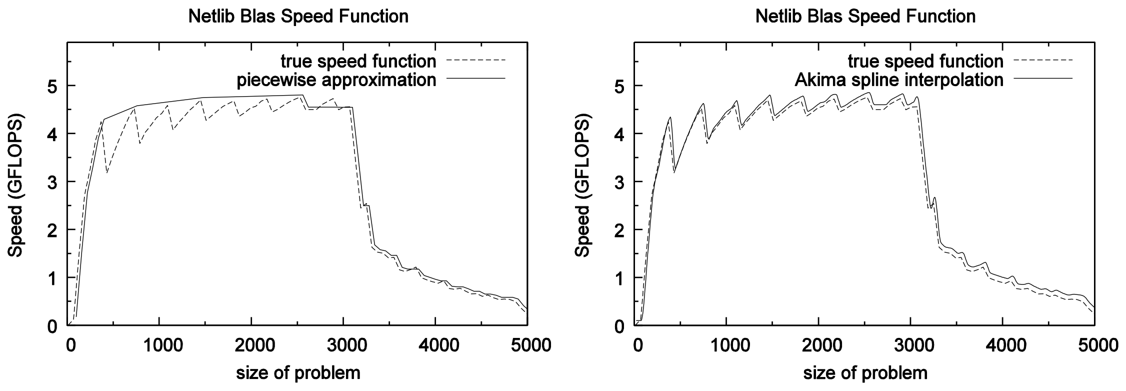


Figure 6. Speed function for non-optimized Netlib BLAS: the piecewise approximation satisfying the restriction of monotonicity (left), and the Akima spline interpolation (right)

These algorithms have been successfully employed in different data-parallel kernels and applications and significantly outperformed their CPM-based counterparts [2, 15, 16, 18, 25, 34].

Algorithms that require full FPMs as input to find the optimal partitioning can be used in applications developed for execution on the same stable platform multiple times. In this case, the cost of building the FPMs for the full range of problem sizes will be insignificant in comparison with the accumulated gains due to the optimal parallelization. However, these algorithms cannot be employed in self-adaptable applications that are supposed to discover the performance characteristics of the executing heterogeneous platform at run-time. To address that type of application, a new class of partitioning algorithms was proposed [36]. They do not need the FPMs as input. Instead, they run on the processors executing the application and iteratively build partial approximations of their speed functions until they become sufficiently accurate to partition the task of the given size with the required precision. For example, if we want to distribute n units of computation between p heterogeneous processors using the geometrical data partitioning, but the speed functions $s_i(x)$ of the processors are not known a priori, we will proceed as follows. The first approximations of the partial speed functions, $\bar{s}_i(x)$, are created as constants $\bar{s}_i(x) = s_i^0 = s_i(n/p)$ as illustrated in Fig. 7(a). At the iteration k, the piecewise linear approximations $\bar{s}_i(x)$ are improved by adding the points (d_i^k, s_i^k), Fig. 7(b). Namely, let $\{(d_i^{(j)}, s_i^{(j)})\}_{j=1}^m$, $d_i^{(1)} < \ldots < d_i^{(m)}$, be the experimentally obtained points of $\bar{s}_i(x)$ used to build its current piecewise linear approximation, then

1. If $d_i^k < d_i^{(1)}$, then the line segment $(0, s_i^{(1)}) \rightarrow (d_i^{(1)}, s_i^{(1)})$ of the $\bar{s}_i(x)$ approximation will be replaced by two connected line segments $(0, s_i^k)) \rightarrow (d_i^k, s_i^k)$ and $(d_i^k, s_i^k) \rightarrow (d_i^{(1)}, s_i^{(1)})$;
2. If $d_i^k > d_i^{(m)}$, then the line $(d_i^{(m)}, s_i^{(m)}) \rightarrow (\infty, s_i^{(m)})$ of this approximation will be replaced by the line segment $(d_i^{(m)}, s_i^{(m)}) \rightarrow (d_i^k, s_i^k)$ and the line $(d_i^k, s_i^k) \rightarrow (\infty, s_i^k)$;

3. If $d_i^{(j)} < d_i^k < d_i^{(j+1)}$, the line segment $(d_i^{(j)}, s_i^{(j)}) \rightarrow (d_i^{(j+1)}, s_i^{(j+1)})$ of $\bar{s}_i(d)$ will be replaced by two connected line segments $(d_i^{(j)}, s_i^{(j)}) \rightarrow (d_i^k, s_i^k)$ and $(d_i^k, s_i^k) \rightarrow (d_i^{(j+1)}, s_i^{(j+1)})$.

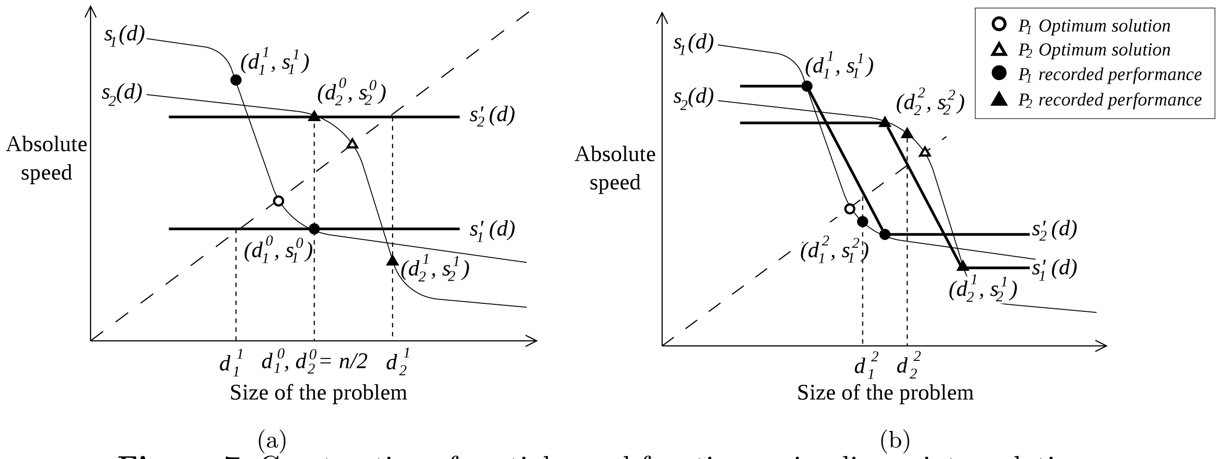


Figure 7. Construction of partial speed functions using linear interpolation.

After adding the new data point (d_i^j, s_i^j) to the partial speed function $\bar{s}_i(x)$, we verify that the shape of the resulting piecewise linear approximation satisfies the above assumptions, and update the value of s_i^j when required. Namely, to keep the partial speed function increasing and convex on the interval $[0, X]$, we ensure that $s_i^{j-1} \leq s_i^j \leq s_i^{j+1}$ and $\frac{s_i^{j-1} - s_i^{j-2}}{d_i^{j-1} - d_i^{j-2}} \geq \frac{s_i^j - s_i^{j-1}}{d_i^j - d_i^{j-1}} \geq \frac{s_i^{j+1} - s_i^j}{d_i^{j+1} - d_i^j}$. The latter expression represents non-increasing tangent of the pieces, which is required for the convex shape of the piecewise linear approximation. On the interval $[X, \infty]$, we ensure that $s_i^{j-1} \geq s_i^j \geq s_i^{j+1}$ for monotonously decreasing speed function.

This approach has proved to be very efficient in practice, typically converging to the optimal solution after a very few iterations [16].

While some other non-constant performance models of heterogeneous processors such as the unit-step functional model [22], the functional model with limits on task size [32] and the band model [30] have been proposed and used for the design of heterogeneous algorithms, they did not go beyond some preliminary studies as they appeared to be not suitable for practical use in high-performance heterogeneous scientific computing due to a variety of reasons.

1.3. Implementation of heterogeneous data partitioning algorithms

It is important to note that the effectiveness of the data partitioning algorithms presented in this section strongly depends on how accurately the performance models employed in these algorithms are reflecting the real performance of the data parallel applications on the executing platforms. Unfortunately many algorithms, especially CPM–based, come without a method for estimation of the employed performance model, leaving this task to the user. Therefore the use of these algorithms as well as tools straightforwardly employing these algorithms is a challenging task. The graph partitioning libraries [4, 11, 12, 28, 45] give us examples of such tools.

At the same time, some algorithm designers include the method of construction of the employed performance model in the definition of the algorithm. Such algorithms are easy to use and compare. The estimation method helps to understand: (i) the meaning of the model parameters leaving no room for interpretation, and (ii) the assumptions made about the application and the target platform better. According to this approach, model-based algorithms will be different even if they only differ in the method of model construction. Such algorithms can be found in [15, 16, 35, 43]. For example, [15] proposes a two-dimensional matrix partitioning algorithm designed for heterogeneous SUMMA (see Fig. 1). The definition of this algorithm specifically stipulates that the FPMs of the processors will be built using the computational kernel performing one update of the submatrix C_i with the portions of pivot block column A_i and pivot block row B_i: $C_i + = A_i \times B_i$ as shown in Fig. 8.

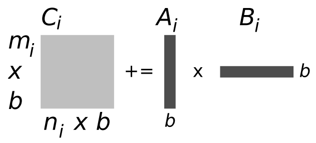


Figure 8. The computational kernel

Moreover, it proposes to use one-dimensional FPMs by combining the height m_i and width n_i parameters into one parameter, area $d_i = m_i \times n_i$, measured in $b \times b$ blocks, and to only use square areas in benchmarking, $m = n = \sqrt{d}$, for $0 < d \leq M \times N$. Then it is partitioned using a one-dimensional FPM-based algorithm to determine the areas of the rectangles that should be partitioned to each processor. The CPM-based algorithm [8] is then applied to calculate the optimum shape and ordering of the rectangles so that the total volume of communication is minimized.

The algorithm described above makes the assumption that a benchmark of a square area will give an accurate prediction of computation time of any rectangle of the same area, namely $s(x, x) = s(x/c, c.x)$. However, in general this does not hold true for all c (Fig. 9(a)). Fortunately, in order to minimise the total volume of communication the algorithm [8] arranges the rectangles so that they are as square as possible. It has been verified experimentally [15] by partitioning a medium sized square dense matrix using the new algorithm for 1 to 1000 nodes from the Grid'5000 platform (incorporating 20 unique nodes), and plotted the frequency of the ratio $m : n$ in Fig. 9(c). Fig. 9(b), showing a detail of Fig. 9(a), illustrates that if the rectangle is approximately square the assumption holds.

The efficiency of the FPM-based data-parallel applications strongly depends on the accuracy of the evaluation of the speed function of each heterogeneous processor. It is a challenging problem that requires: (i) carefully designed experiments to accurately and efficiently measure the speed of the processor for each problem size; (ii) appropriate interpolation and approximation methods which use the experimental points to construct an accurate speed function of the given shape. A software tool, FuPerMod, helping the application programmer solve these problems has been recently developed and released [17]. FuPerMod also provides a number of heterogeneous data partitioning algorithms for sets, ordered sets and matrices, both CPM-based and FPM-based. It does not provide graph-partitioning algorithms though. Graph-partitioning algorithms are provided by a number of libraries such as ParMetis [28], SCOTCH [12], JOSTLE [45], Zoltan [11], PaGrid [4]. While the partitioning algorithms implemented in these libraries use performance models, the libraries provide no support for their construction.

2. Optimization of parallel applications on hybrid multicore and multi-accelerator heterogeneous platforms

Thus, the traditional heterogeneous performance models and data partitioning algorithms and applications are designed for platforms whose processing elements are independent of each

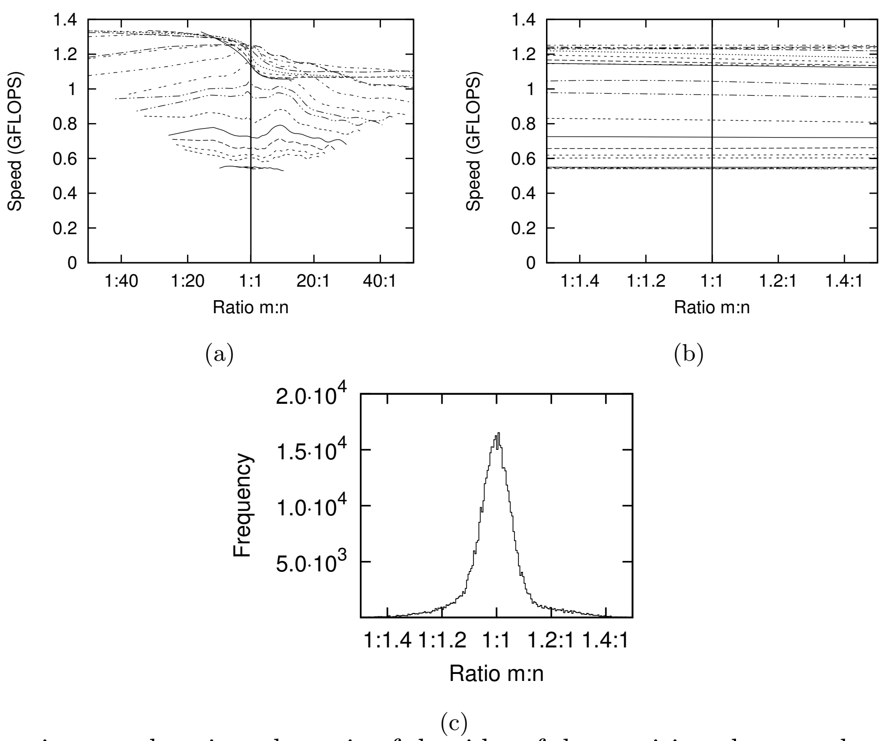


Figure 9. Showing speed against the ratio of the sides of the partitioned rectangles. Lines connect rectangles of equal area. The centerline at $1:1$ represents square shape. In general speed is not constant with area (a). However when the ratio is close to $1:1$, speed is approximately constant (b). (c) Shows the frequency distribution of the ratio of $m:n$ using the new partitioning algorithm for 1 to 1000 machines (incorporating 20 unique hardware configurations)

other. In modern heterogeneous multicore and multi-accelerator compute nodes, however, processing elements are coupled and share system resources. In such platforms, the speed of one processing element often depends on the load of others due to resource contention. Therefore, they cannot be considered independent, and hence their associated performance models cannot be considered and built independently. This makes the traditional models, methods of their evaluation and algorithms no longer applicable to the new platforms.

This problem was recently addressed in [46] [47] [48]. In this work, the authors do not study how to develop computational kernels for individual computing devices used in hybrid heterogeneous platforms, such as multicore CPUs or GPUs. They assume that such kernels are available for the use in parallel applications on these platforms. While being very challenging and important, this problem has attracted significant attention of the HPC research community and many important kernels have been ported to modern multicores and GPUs. Instead, they focus on a wide open problem of optimal data distribution between kernels of the data-parallel application assuming that the configuration of the application is fixed. Finding the optimal configuration of the application is another challenge to be addressed, which is out of the scope of this work. The authors however give few basic empirical rules that, they believe, lead to optimal configurations. For example, never run a NUMA-unaware multi-threaded computational kernel across multiple NUMA nodes. Use instead multiple instances of this kernel, one per NUMA node.

A multicore and multi-GPU system, which the main target architecture in this work, is modeled by a set of heterogeneous abstract processors determined by the configuration of the parallel application. Namely, a group of processing elements executing one computational kernel of the application will make a combined processing unit and will be represented in the model by one abstract processor. For example, if a single-threaded computational kernel is used, then each CPU core executing this kernel will be represented in the model by a separate abstract processor. If a multi-threaded computational kernel is used, then each group of CPU cores executing the kernel will make a combined processing unit represented in the model by one abstract processor. A GPU is usually controlled by a host process running on a dedicated CPU core. This process instructs the GPU to perform computations and handles data transfers between the host and device memory. In the case of a single-GPU computational kernel, the GPU and its dedicated CPU core will make a combined processing unit represented by an abstract processor. If a multi-GPU computational kernel is used in the application, the GPUs and their dedicated CPU core will make a combined processing unit represented by one abstract processor.

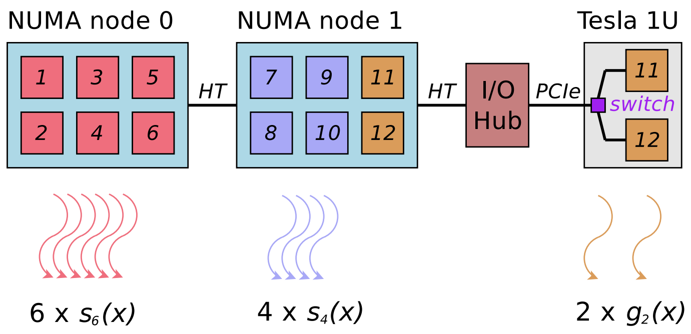


Figure 10. Performance modeling on a GPU-accelerated multicore server of NUMA architecture: single-threaded and single-GPU computational kernels executed

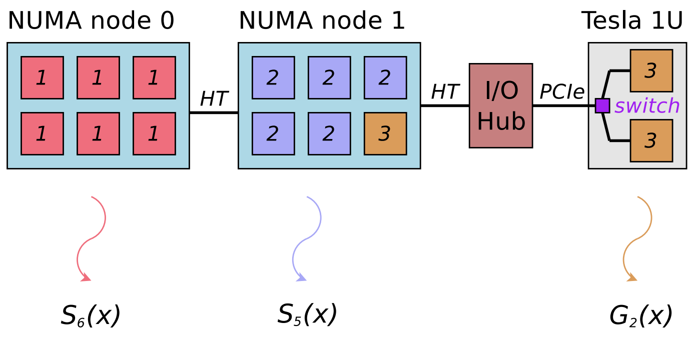


Figure 11. Performance modeling on a GPU-accelerated multicore server of NUMA architecture: multi-threaded and multi-GPU computational kernels executed; two GPUs handled by a single dedicated CPU core

Figures 10 and 11 illustrate this approach showing a GPU-accelerated multicore server of NUMA architecture executing a parallel application in two different configurations. The configuration shown in Fig. 10 is based on the single-threaded and single-GPU computational kernels. It consists of ten processes running the CPU kernels on ten cores of both NUMA nodes, and two processes running the GPU kernels on accelerators and their dedicated cores on the second NUMA node. The configuration in Fig. 11 is based on the multi-threaded and multi-GPU computational kernels. It consists of one process running the 6-thread CPU kernel on one NUMA node, one process running the 5-thread CPU kernel on another NUMA node, and one process running the GPU kernel on the GPUs and their single dedicated core. All processing elements in these diagrams are enumerated. Each number indicates the combined processing unit to which the processing element belongs. For example, in the first configuration, the cores in NUMA node 0 make six processing units, and each GPU with its dedicated CPU core in NUMA node 1 make a combined processing unit.

In the first configuration, the cores in NUMA node 0 execute six identical processes and are modeled by six abstract processors. These cores are tightly coupled and share memory, therefore, they cannot be considered independent. On the other hand, this group of processing elements is relatively independent of other processing elements of the server. Therefore, their performance should be measured simultaneously in a group but can be measured separately from the others. In the second configuration, these six cores execute one process and modeled as one combined processing unit. Its performance can be measured separately from other processing elements of the server.

Next steps are to build functional performance models of the abstract processors and perform model-based data partitioning in order to balance the workload between the combined processing units represented by these abstract processors.

In order to build the performance models of the abstract processors, the performance of the processing units representing these processors has to be measured. To measure the performance of the processing units accurately, they are grouped by the shared system resources, so that the resources be shared within each group but not shared between groups. The performance of processing units in a group is measured when all processing units in the group are executing some workload simultaneously, thereby taking into account the influence of resource contention. To prevent the operating system from migrating processes excessively, processes are bound to CPU cores. Processes are synchronized to minimize the idle computational cycles, aiming at the highest floating point rate for the application. Synchronization also ensures that the resources will be shared between the maximum number of processes. To ensure the reliability of the results, measurements are repeated multiple times, and average execution times are used.

One important empirical rule used in this work is that when looking for the optimal distribution of the workload, only the solutions that evenly distribute the workload between identical CPU processing units are considered. This simplification significantly reduces the complexity of the data partitioning problem. It is based both on the authors' extensive experiments that have shown no evidence that uneven distribution between identical processing units could speed up applications, and on the absence of such evidence in literature. Therefore, identical processing units that share system resources will be always given the same amount of workload during performance measurements.

To account for different configurations of the application, three types of functional performance models for CPU cores are defined:

1. $s(x)$ approximates the speed of a uniprocessor executing a single-threaded computational kernel. The speed $s(x) = x/t$, where x is the number of computation units, and t is the execution time.
2. $s_c(x)$ approximates the speed of one of c CPU cores all executing the same single-threaded computational kernel simultaneously. The speed $s_c(x) = x/t$, where x is the number of computation units executed by each CPU core, and t is the execution time.
3. $S_c(x)$ approximates the collective speed of c CPU cores executing a multi-threaded computational kernel. The speed $S_c(x) = x/t$, where x is the total number of computation units executed by all c CPU cores, and t is the execution time. $S_c(cx)/c$ is used to approximate the average speed of a CPU core.

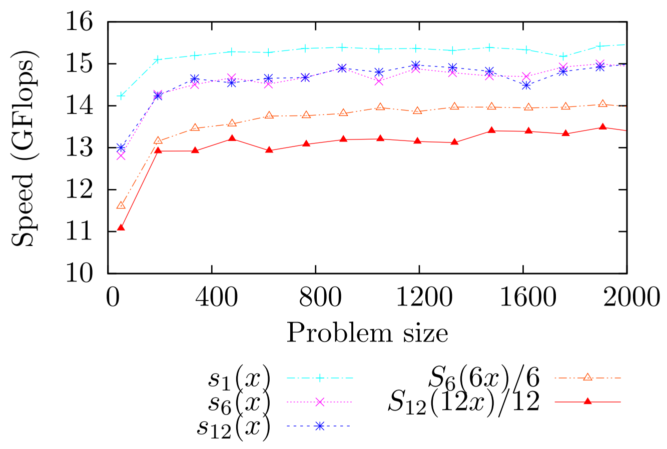


Figure 12. Speed functions of a CPU core built in different configurations

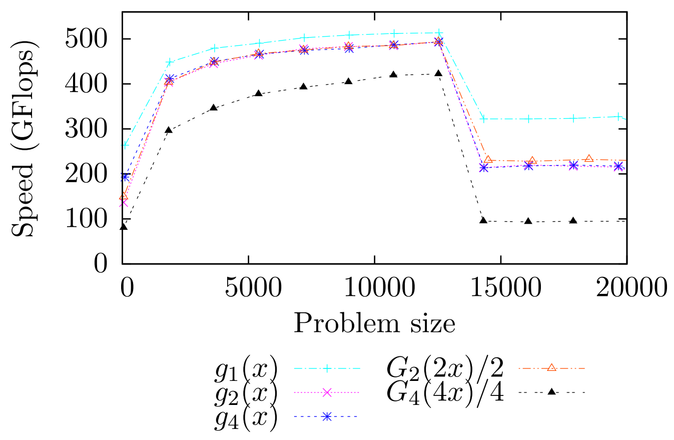


Figure 13. Speed functions of a GPU processing unit built in different configurations

Fig. 12 shows speed functions of a CPU core built in different configurations on a server, consisting of eight NUMA nodes connected by AMD HyperTransport(HT) links, with 6 cores and 16 GB local memory each. The server is equipped with a NVIDIA Tesla S2050 server, which consists of two pairs of GPUs. Each pair is connected by a PCIe switch and linked to a separate NUMA node by a PCIe bus.

Similarly, three types of functional performance models for GPUs are defined as follows:

1. $g(x)$ approximates the speed of a combined processing unit made of a GPU and its dedicated CPU core that execute a single-GPU computational kernel, exclusively using a PCIe link. The speed $g(x) = x/t$, where x is the number of computation units, and t is the execution time.
2. $g_d(x)$ approximates the speed of one of d combined processing units, each made of a GPU and its dedicated CPU core. All processing units execute identical single-GPU computational kernels simultaneously. The speed $g_d(x) = x/t$, where x is the number of computation units executed by each GPU processing unit, and t is the execution time.
3. $G_d(x)$ approximates the speed of a combined processing unit made of d GPUs and their dedicated CPU core that collectively execute a multi-GPU computational kernel. The speed $G_d(x) = x/t$, where x is the total number of computation units processed by all d GPUs, and t is the execution time. $G_d(dx)/d$ is used to approximate the average speed of a GPU.

Fig. 13 shows the speed functions of a combined GPU processing unit built in different configurations on the same server.

From these experiments we can see that depending on the configuration of the application the speed of individual cores and GPUs can vary significantly. Therefore, to achieve optimal distribution of computations it is very important to build and use speed functions which accurately reflect their performance during the execution of the application. This work also reveals that the speed of GPU can depend on the load of CPU cores, which should be also taken into account during the partitioning step. Experiments with linear algebra kernels and a CFD application validated the efficiency of the proposed approach.

At the same time, this work has demonstrated the importance of proper configuration of the application. For example, Fig. 14 demonstrates the impact of NUMA mapping on the performance of a GPU processing unit, comprised of a CPU core and a GPU of Tesla S2050 deployed in the experimental server. $g_1(x)$ is built by executing one single-GPU *gemm* kernel, which uses exclusively the data link and the memory of a local or remote NUMA node. $g_2(x)$ is built by executing two single-GPU kernels simultaneously on two GPU units that share the PCIe link and the memory of the same NUMA node, local or remote. In the remote configuration, the GPU units also share an extra HT link to the remote NUMA node. Speed function $g_2(x)$ is also built in the configuration when two dedicated CPU cores are located on different NUMA nodes, which is denoted as *local + remote*. In this case, the processing units share PCIe but do not share memory.

The difference between speed functions $g_1(x)$ and $g_2(x)$ reflects the performance degradation due to the contention for PCIe, HT and memory. Significant difference is observed for large problem sizes when many data transfers are required. Communication overhead between NUMA nodes can be estimated by the difference between $g_1(x)$ in *local* and *remote* configurations. The combined effect of both phenomena is reflected by the $g_2(x)$ functions in different configurations.

Multilevel hierarchy in modern heterogeneous clusters represents another challenge to be addressed in the design of data partitioning algorithms. One solution, a hierarchical matrix partitioning algorithm based on realistic performance models at each level of hierarchy, was recently proposed in [14]. To minimize the total execution time of the application it iteratively partitions a matrix between nodes and partitions these sub-matrices between the devices in a node. This is a self-adaptive algorithm that dynamically builds the performance models at run-time and it employs an algorithm to minimize the total volume of communication. This algorithm

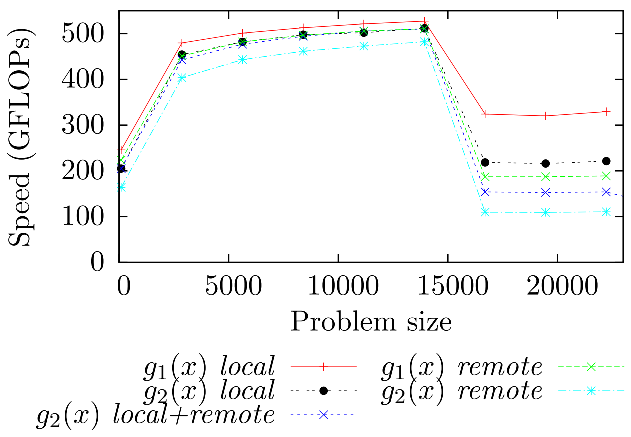


Figure 14. Speed functions of a GPU processing unit built in different configurations

allows scientific applications to perform load balanced matrix operations with nested parallelism on hierarchical heterogeneous platforms. Large scale experiments on a heterogeneous multi-cluster site incorporating multicore CPUs and GPU nodes have shown that this hierarchical algorithm outperforms all other state–of–the–art approaches and successfully load balance very large problems.

3. Programming tools

In the past, the main platform used for non-trivial heterogeneous parallel computing (as opposed to volunteer computing, such as the seti@home project) has been a heterogeneous cluster of workstations. MPI is a standard programming model for this platform. However, the implementation of real-world heterogeneous parallel algorithms in an efficient and portable form requires much more than just the code implementing the algorithm for each legal combination of its input parameters. Extra code should be written to find optimal values of some parameters (say, the number of processes and their arrangement in a multi-dimensional shape) or to accurately estimate the others (such as relative speeds of the processors). This extra code may account for at least 95% of all code in common cases. Therefore, for the implementation of heterogeneous parallel algorithms on this platform, a small number of programming tools was developed. mpC [3] is the first programming language designed for heterogeneous parallel computing. It facilitates the implementation of heterogeneous parallel algorithms by automating the development of the routine code, which comes in two forms: (i) application specific code generated by a compiler from the specification of the implemented algorithm provided by the application programmer; (ii) universal code in the form of run-time support system and libraries. HeteroMPI [33] is an extension of MPI inspired by mpC. It allows the programmer to re-use the available MPI code when developing applications for heterogeneous clusters of workstations. Both mpC and HeteroMPI have been used for development of a wide range of real-life applications. HeteroMPI was also the instrumental tool for implementation of Heterogeneous ScaLAPACK [42], a version of ScaLAPACK optimized for heterogeneous clusters of workstations.

Modern and future heterogeneous HPC systems necessitate the synthesis of multiple programming models in the same code. This will be a result of the use of multiple heterogeneous many-core devices for accelerating code, as well as the use of both shared- and distributed-address spaces in the same code to cope with heterogeneous memory hierarchies and forms of communication. Synthesizing multiple programming models in the same code in a way that would provide a good balance of performance, portability and programmability, is far from trivial. Despite long-standing efforts to program parallel applications with hybrid programming models (e.g. MPI/OpenMP) and some recent developments in programming models for hybrid architectures (e.g. OpenCL), it is still a long way towards solutions that would satisfy the HPC community.

This work was conducted with the financial support of Science Foundation Ireland, Grant 08/IN.1/I2054.

References

1. H. Akima. A new method of interpolation and smooth curve fitting based on local procedures. *Journal of the ACM*, 17:589–602, 1970.
2. A. Alonazi, D. Keyes, A. Lastovetsky, and V. Rychkov. Design and optimization of openfoam-based cfd applications for hybrid and heterogeneous hpc platforms. 26th International Conference on Parallel Computational Fluid Dynamics (ParCFD 2014), Trondheim, Norway, 2014.
3. D. Arapov, A. Kalinov, A. Lastovetsky, I. Ledovskih, and T. Lewis. A programming environment for heterogenous distributed memory machines. In *6th Heterogeneous Computing Workshop (HCW 1997)*, pages 32–45. IEEE, 1997.
4. E. Aubanel and X. Wu. Incorporating latency in heterogeneous graph partitioning. In *IPDPS 2007*, pages 1–8, 2007.
5. C. Augonnet et al. Automatic calibration of performance models on heterogeneous multicore architectures. In *EuroPar*, 2009.
6. J. Barbosa, J. Tavares, and A. J. Padilha. Linear algebra algorithms in a heterogeneous cluster of personal computers. In *9th Heterogeneous Computing Workshop (HCW 2000)*, pages 147–159, 2000.
7. O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert. A proposal for a heterogeneous cluster scalapack (dense linear solvers). *IEEE Transactions on Computers*, 50(10):1052–1070, 2001.
8. O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.
9. R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.
10. P. Boulet, J. Dongarra, F. Rastello, Y. Robert, and F. Vivien. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters*, 9(2):197–213, 1999.
11. U. Catalyurek, E. Boman, K. Devine, et al. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *IPDPS 2007*, pages 1–11, 2007.
12. C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6–8):318–331, 2008.
13. J. Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *HPC Asia*, pages 224 –229, 1997.
14. D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa. Hierarchical partitioning algorithm for scientific computing on highly heterogeneous cpu+ gpu clusters. In *Euro-Par 2012 Parallel Processing*, pages 489–501. Springer, 2012.
15. D. Clarke, A. Lastovetsky, and V. Rychkov. Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models. In *HeteroPar 2011*, pages 450–459. Springer, 2011.
16. D. Clarke, A. Lastovetsky, and V. Rychkov. Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms. *Parallel Processing Letters*, 21(2):195–217, 2011.

17. D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky. Fupermod: A framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms. In *PaCT 2013*, volume 7979 of *LNCS*, pages 182–196. Springer, 2013.

18. J. Colaco, A. Matoga, et al. Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems. In *PPAM 2013, Part I*, pages 693–703, 2014.

19. A. DeFlumere and A. Lastovetsky. Searching for the optimal data partitioning shape for parallel matrix matrix multiplication on 3 heterogeneous processors. In *23rd Heterogeneity in Computing Workshop (HCW 2014)*, pages 1–12, 2014.

20. A. DeFlumere, A. Lastovetsky, and B. Becker. Partitioning for parallel matrix multiplication with heterogeneous processors: The optimal solution. In *21st Heterogeneity in Computing Workshop (HCW 2012)*, pages 1–15, 2012.

21. K. Dichev and A. Lastovetsky. Optimization of collective communication for heterogeneous hpc platforms. *High-Performance Computing on Complex Environments*, pages 95–114, 2014.

22. M. Drozdowski and P. Wolniewicz. Out-of-core divisible load processing. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1048–1056, 2003.

23. R. Van De Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

24. R. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Computing*, 20(3):389–398, 1994.

25. A. Ilic, F. Pratas, P. Trancoso, and L. Sousa. High-performance computing on heterogeneous systems: Database queries on cpu and gpu. In *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives.* IOS Press, 2011.

26. A. Ilic and L. Sousa. On realistic divisible load scheduling in highly heterogeneous distributed systems. In *PDP 2012*, pages 426–433. IEEE, 2012.

27. A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In *7th International Conference on High Performance Computing and Networking Europe (HPCN'99)*, pages 191–200, 1999.

28. G. Karypis and K. Schloegel. *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version 4.0.* University of Minnesota, MN, USA, 2013.

29. A. Lastovetsky. On grid-based matrix partitioning for heterogeneous processors. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, pages 383–390. IEEE, 2007.

30. A. Lastovetsky and R. Higgins. Scheduling for heterogeneous networks of computers with persistent fluctuation of load. In *13th International Conference on Parallel Computing (ParCo 2005)*, pages 383–390, 2005.

31. A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *IPDPS 2004*, pages 1–15, 2004.

32. A. Lastovetsky and R. Reddy. Data partitioning for multiprocessors with memory heterogeneity and memory constraints. *Scientific Programming*, 13(2):93–112, 2005.

33. A. Lastovetsky and R. Reddy. Heterompi: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.
34. A. Lastovetsky and R. Reddy. Data partitioning for dense factorization on computers with memory heterogeneity. *Parallel Computing*, 33(12):757–779, 2007.
35. A. Lastovetsky and R. Reddy. Data partitioning with a functional performance model of heterogeneous processors. *International Journal of High Performance Computing Applications*, 21:76–90, 2007.
36. A. Lastovetsky and R. Reddy. Distributed data partitioning for heterogeneous processors based on partial estimation of their functional performance models. In *Euro-Par'09*, pages 91–101, 2009.
37. A. Lastovetsky, R. Reddy, and R. Higgins. Building the functional performance model of a processor. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, pages 746–753. ACM, 2006.
38. A. Lastovetsky and J. Twamley. Towards a realistic performance model for networks of heterogeneous computers. In *High Performance Computational Science and Engineering*, pages 39–57. Springer, 2005.
39. M. Linderman, J. Collins, H. Wang, et al. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43:287–296, 2008.
40. G. Quintana-Ortí et al. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 44:121–130, 2009.
41. J.N. Quintin and F. Wagner. Hierarchical work-stealing. *Euro-Par 2010-Parallel Processing*, pages 217–229, 2010.
42. R. Reddy and A. Lastovetsky. Heterompi+ scalapack: towards a scalapack (dense linear solvers) on heterogeneous networks of computers. In *13th IEEE International Conference on High Performance Computing (HiPC 2006)*, pages 242–253. 2006.
43. V. Rychkov, D. Clarke, and A. Lastovetsky. Using multidimensional solvers for optimal data partitioning on dedicated heterogeneous hpc platforms. In *PaCT 2011*, pages 332–346. Springer-Verlag, 2011.
44. F. Song et al. Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems. In *ICS*, 2012.
45. C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Computer Systems*, 17(5):601–623, 2001.
46. Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore platforms. In *Cluster 2011*, pages 580–584. IEEE, 2011.
47. Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications. In *Cluster 2012*, pages 191–199. IEEE, 2012.
48. Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore and multi-gpu platforms using functional performance models. *IEEE Transactions on Computers*, pages 1–14, 2014.

DOI: 10.14529/jsfi140305

Co-design of Parallel Numerical Methods for Plasma Physics and Astrophysics

Boris M. Glinskiy[1,2], ***Igor M. Kulikov***[1,2,3], ***Alexey V. Snytnikov***[1,2], ***Alexey A. Romanenko***[2], ***Igor G. Chernykh***[1,2], ***Vitaly A. Vshivkov***[1,2]

Physically meaningful simulations in plasma physics and astrophysics need powerful hybrid supercomputers equipped with computation accelerators. The development of parallel numerical codes for such supercomputers is a complex scientific problem. In order to solve it the concept of co-design is employed. The co-design is defined as considering the architecture of the supercomputer at all stages of the development of the code. The use of co-design is shown by the example of two physical problems: the interaction of an electron beam with plasma and the collision of galaxies. The efficiency is 92 % with 500 Tesla GPUs at the Lomonosov supercomputer. The test computation involved 160 million of model particles.

Keywords: Co-design, hybrid supercomputers, Particle-In-Cell method, Godunov method, GPU.

Introduction

The main question in plasma simulation is the correct computation of the interaction of plasma particles with the waves in plasma. The waves are the basis of plasma instabilities. The correct simulation of plasma instabilities requires at least 16 grid nodes at the Debye length. The number of model particles cannot be small (less than 100 per cell). The simulations with small number of particles lead to poor quality results in well-known physical situations and to the lack of understanding in the new physical problems [1]. Moreover, the simulation of plasma instabilities is essentially a 3D probem.

The particular physical problem is the simulation of Langmuir waves excitation by the relativistic electron beam with plasma. One must notice that the simulation of the beam-plasma interaction is important for a lot of other plasma physics problems.

Beam-plasma interaction plays an important role in various physical phenomena, such as the transport of relativistic electrons in fast ignition scheme within inertial fusion, the gamma-bursts, solar radio bursts of II and III type, also in formation of collisionless shock waves in space plasma (see review [2] and references therein). Moreover, the collective processes in the beam-plasma system derermine the efficiency of turbulent plasma heating [3, 4] and also of electromagnetic radiation generation [5–8] in mirror traps.

The movement of galaxies in dense clusters turns the collisions of galaxies into an important evolutionary factor, because during the Hubble time an ordinary galaxy may suffer up to 10 collisions with the galaxies of its cluster. Observational and theoretical study of interacting galaxies is an indispensable method for studying their properties and evolution.

One of the most important computational problems within the study of galaxies is the ratio of the characteristics lengths. The size of a galaxy is 10^4 parsec, and the size of a star is about 10^{-7} parsec. Thus the solution of such problems requires the supercomputers from the top part of the Top500 list. Two of the first three (five of the first ten) supercomputers are equipped

[1]Institute of Computational Mathematics and Mathematical Geophysics, Novosibirsk, Russia
[2]Novosibirsk State University, Novosibirsk, Russia
[3]Novosibirsk State Technical University, Novosibirsk, Russia

with computation accelerators (either Nvidia GPUs of Intel Xeon Phi) in the most recent (June 2014) Top500 list.

The first Exaflops supercomputer will be most likely built on the basis of some computation accelerators. At present there are already codes for plasma physics adopted for both GPUs and Intel Xeon Phi accelerators [10, 11]. Still, the design of the codes for the hybrid supercomputers (i.e. the supercomputers equipped with accelerators) is not just a technical question. On the contrary, it is a complex scientific problem. The problem requires co-design of the algorithms at all stages of the solution of the problem from the physical statement of the problem to the software development tools. Within the numerical simulation context co-design is the design of physical and mathematical model, of the numerical method, of the parallel algorithm and of the implementation using the architecure of the hybrid supercomputer efficiently.

The need for top supercomputers in plasma physics exist because of the recent trend of using more precise models instead of simplified ones such as hydrodynamical. The more precise models are based on the Vlasov equation. The Vlasov equation is solved either by the explicit numerical methods in the 6D space of by the Particle-In-Cell (PIC) method [12]. If the PIC method is used then the quantitatively correct physical result may be obtained only with the large number of model particles(from 1 to 10 thousands per cell).

The simulations were conducted of the relativistic electron beam interacting with plasma [32]. These simulations resulted in the correct value of the two-stream instability increment. In order to obtain the correct value the number of model particles was increased up to 1000 per cell. The number of grid nodes is $120 \times 4 \times 4$, domain size is 1.2 (in non-dimensional units). The solution time is about 3 days with 256 Intel Xeon cores (64 4-core Intel Xeon processors). The full 3D simulation will need at least 100 nodes in both transverse dimensions and several times more nodes in the longitudinal dimension keeping the number of model particles per cell, resulting in hundreds of millions of model particles as a whole. It means the increase of computational workload in several thousand times. Let us use 5000. Thus, in order to keep the wallclock time of the simulation (3 days) one has to use 256×5000=1.2 million cores!!! It is the size of the IBM BlueGene/Q, the present number 3. But since it is just one single simulation of at least 10 usually needed to solve a physical problem, it is strongly necessary to do such simulations with lower number of processors and less electric power. That is why using hybrid supercomputers for plasma simulation is inevitable. This requires co-design of the parallel algorithms.

In recent two decades two approaches are being mainly used for the solution of non-stationary 3D astrophysical problems. First, it is the Lagrangian approach mainly represented by SPH method [13, 14] (Smoothed Particle Hydrodynamics) and Eulerian approach with adaptive meshes, or AMR [15] (Adaptive Mesh Refinement). Within the Lagrangian approach the following codes were developed: Hydra [16], Gasoline [17], GrapeSPH [18], GADGET [19] on the basis on SPH method. Within Eulerian approach the following codes were developed: NIRVANA [20], FLASH [21], ZEUS [22], ENZO [15], RAMSES [23], ART [24], Athena [25], Pencil Code [26], Heracles [27], Orion [28], Pluto [29], CASTRO [30], GAMER [31].

The statement of the problem for the galactic objects dynamics is the solution of the equation of the gravitational magnetic gas dynamics for the gas component considering magnetic field and self-gravity and also the solution of the N-body problem for the collisionless component. It is well-known that the N-body problem is hard to solve with supercomputers especially with hybrid architecture. Thus it is necessary to find a method to describe the collisionless component, the method being suitable for efficient parallel implementation. In [33] such a model was developed on the basis of the first moment of the Boltzmann equation. Such an approach makes it possible to use the same numerical algorithm [34] for the solution of magnetic gas dynamics equations, and also for the solution of the first moments of the Boltzmann equation [35]. This approach was efficiently implemented for two types of hybrid supercomputers: for the supercomputer with GPUs [35], and the supercomputer with Intel Xeon Phi accelerators [11].

1. Co-design of parallel numerical methods

The essence of co-design is the analysis of the efficiency of the parallel implementation at all stages of the development of the algorithm. The stages are the following:

- Physical consideration of the model
- Mathematical statement of the problem: differential equations
- Numerical method
- Parallel algorithm
- Architecture of the supercomputer
- Parallel programming tools

1.1. Co-design of parallel numerical methods for plasma physics

In the present case co-design begins at the stage of the physical consideration of the problem. It is known from experiment that the plasma density modulation cannot exceed 300 %. It means that the number of model particles cannot be increased without a limit. Thus there is no need for dynamic load balancing, and the absence of dynamic balancing improves the reliability and efficiency of the parallel implementation.

At the stage of the numerical method design the field evaluation method was chosen that is built on the basis of Faraday and Ampere laws. In such a way there is no need to solve Poisson equation. Instead, the equations that represent the Faraday and Ampere laws in the numerical form are solved by the Langdon-Lasinski scheme. This results in the field solver with virtually unlimited scalability.

At the stage of supercomputer architecture selection the PIC method details are taken into account. In order to evaluate the new values of position and impulse of a particle it is necessary to know the values of electric and magnetic fields at the present position of the particle. Each of the three components of both electric and magnetic field is stored in a separate 3D array. In such a way six 3D arrays are accessed at each time step for each particle. Since the particles are situated randomly in the domain, the access to the arrays is also unordered. It means that the use of the cache memory can not reduce the computation time. If a part of the field array was fetched to the cache in the process of computation of the particle movement, it would be impossible to use this part of the field array for the computation with the next particle, because it is (most likely) situated in a completely different part of the subdomain. Since the cache memory can not store all the six arrays for fields, one has to access the RAM (Random Access Memory) for computation of the particle movement. And since the performance of the processor is usually limited by the memory bandwidth, it is the memory bandwidth that determines the speed of the computation with particles and the performance of the program as a whole (particles take from 60 % to 90 % of the total time). This fact determines the transition to the supercomputers equipped with GPUs because CUDA has a lot of tools to accelerate memory use for the PIC method.

At the stage selection of software design tools is the folowing. For the PIC method with a very big number of independently processed elements (the model particles) the use of CUDA technology is very efficient. Other parallel technologies for hybrid supercomputers such as OpenCL, OpenMP, OpenACC could be also used but it is CUDA that gives the possibility to use the highest number of parallel processes and to get the highest performance.

The last stage of co-design is the adoptation of the algorithm to the GPU architecture. The traditional PIC method implementation (all the particles stored in one very big array) is unacceptable for GPUs because it prevents the use of the advantages of GPU memory. Thus the particles are distributed between the cells and the computation is conducted as follows: one block of threads for one cell.

1.2. Co-design of parallel numerical methods for astrophysics

In the design of astrophysics models we shall follow the idea of co-design that means the design of parallel numerical technologies considering all the aspects of parallelism. For example, in [33] a new approach was proposed for the simulation of the collisionless component of the galaxies. The approach is based on the first moments of the collisionless Boltzmann equation. This approach has some limitations if it is necessary to trace a single particle and not a group of particles (for example, in the study of planet formation). But for the simulation of interacting galaxies this approach was successfully tested [35]. As it was already said in the introduction, the statement of the problem for the galactic objects dynamics is the solution of the equation of the gravitational magnetic gas dynamics for the gas component considering magnetic field and self-gravity and also the solution of the equations for the first moments of the collisionless Boltzmann equations. During the last two decades for the solution of non-stationary 3D problems

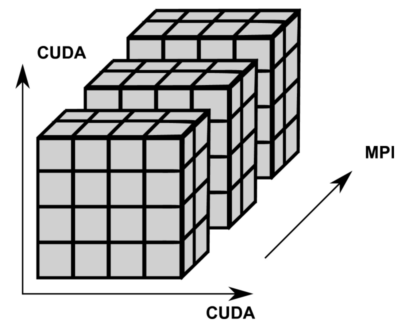


Figure 1. Domain decomposition for GPU-based supercomputers

two approaches are mostly used from the wide range of gas dynamics methods. This is the Lagrangian approach mostly represented by the SPH method and Eulerian approach with the use of adaptive meshes or AMR. The main weak point of SPH is the bad simulation of high gradients and discontinuities, suppression of instabilities, difficult choice of the smoothing kernel and the need for artificial viscosity. A large standalone problem is the local entropy decrease in the SPH method. The main weak point of the mesh-based methods is the non-invariance with respect to rotation, or Galilean non-invariance. This invariance appears because of the mesh. But this problem can be solved by means of the various approaches to the design of numerical schemes [9]. One of the common weak points of both Lagragian and Eulerian approaches is the lack of scalability. In recent time the mixed Lagrangian-Eulerian methods are employed for the solution of astrophysical problems. These methods combine the advantages of both Lagrangian and Eulerian approaches. For a number of years the authors have been developing the Lagrangian-Eulerian approach on the basis of the combination of the Fluid-In-Cells method and Godunov method [9, 11, 34, 35]. The system of gas dynamics equations is solved in two stages. During the first, Eulerian stage, the system is solved without the advection terms. During the second, Lagrangian stage, the advection is taken into account. The separation into two stages facilitates the elimination of the Galilean non-invariance. The use of Godunov method at the Eulerian stage enables the correct simulation of discontinuos solutions.

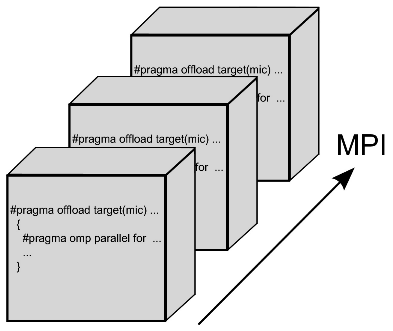

#pragma offload target(mic) ...
for ...
#pragma offload target(mic) ...
for ...
#pragma offload target(mic) ...
{
#pragma omp parallel for ...
...
}
MPI

Figure 2. Domain decomposition for Xeon Phi-based supercomputers

The use of uniform Cartesian grids makes it possible to choose an arbitrary Cartesian domain decomposition. In fig. 1 and fig. 2 the two ways of the domain decomposition for hybrid supercomputers are given: for the GPU-based supercomputer (fig. 1) and for the supercomputer equipped with Intel Xeon Phi (fig. 2).

2. Speedup and efficiency

2.1. Co-design for astrophysics: the codes GPUPEGAS and AstroPhi

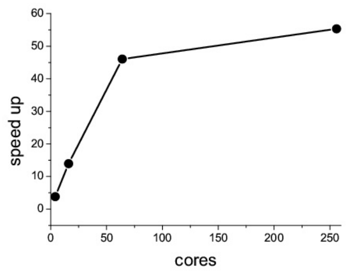


Figure 3. Speedup of the GPUPEGAS code within a single GPU

The use of such domain decomposition enables the most efficient use of the computational accelerators. In fig. 3 and fig. 4 the speedup is given with the single GPU (fig. 3) and a single Intel Xeon Phi accelerator (fig. 4) in the different modes.

With the use of 60 GPUs and 32 Intel Xeon Phi accelerators the parallel efficiency exceeded 95 %. The wallclock time for the AstroPhi code is the following: 1 Xeon core 10,664 sec., 1 Xeon Phi core in offload mode 81,708 sec. (same as native mode), 60 Xeon Phi cores in offload mode 2,960 sec. (28 times speedup), 60 Xeon Phi cores in native mode 1,547 sec. (53 times speedup). For the GPUPEGAS code 1 Xeon core 10,664 sec., 1 GPU core 14,575 sec., 256 GPU cores 0,265 sec. (55 times speedup).

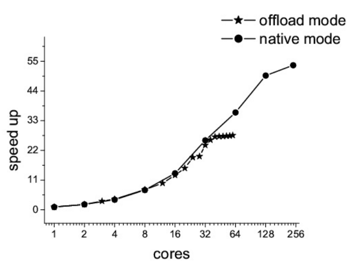


Figure 4. Speedup of the AstroPhi code within a single Intel Xeon Phi

2.2. Co-design for plasma physics: the code for beam-plasma interaction simulation

The use of co-design for the development of the PIC code for plasma simulation resulted in the 42 times speedup with Nvidia Kepler. Speedup here is given with respect to 4-core Intel Xeon processor (all 4 cores employed). The efficiency of the program measured with MVS-100K is over 60 % with up to 2048 Intel Xeon cores (in the figure they are called processors since here each core acts as a stanalone processor, from the MPI point of view). It is shown in fig. 5.

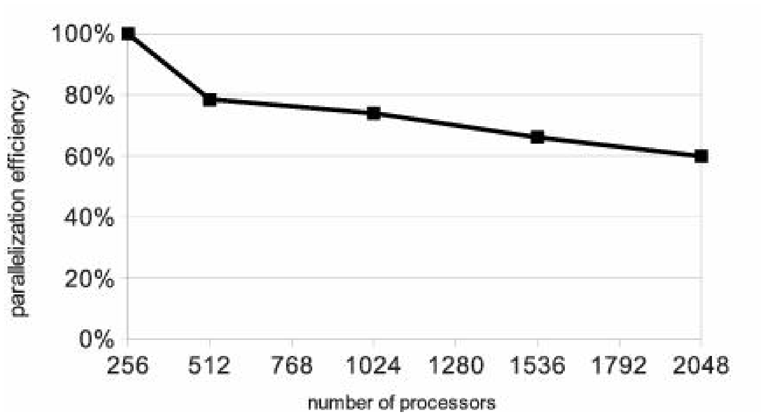


Figure 5. Parallelization efficiency measured with cluster named MVS-100K, Joint Supercomputer Centre of the RAS. The grid size along Y and Z is 64 nodes, grid size along X is equal to the number of processors, 150 particles per cell for all the cases.

The parallel program was developed primarily for the simulation of beam interaction with plasma on large computational grids and with large numbers of particles. It is interesting to know, what will the worktime be for a larger problem. For example, if there is a small problem solved with a small computer in some time, is it possible to solve a larger problem with a larger computer in the same time? Or will the time be larger? How much? That is why parallelization efficiency was computed in the following way.

$$k = \frac{T_2}{T_1} \times 100\% \tag{1}$$

here T_1 - is the computation time with N_1 processors, T_2 is the computation time with N_2 processors. Here the workload of a single processor is constant. This definition of efficiency is inversely proportional to the standard weak scale efficiency, but the above efficiency definition is better since it clearly shows how the things get worse with a lot of processors (with weak scale efficiency one notices that the plot stops growing, but the values are still high, so it seems to be good, though it is not). In the ideal case the computation time must remain the same (the ideal $k = 100\%$).

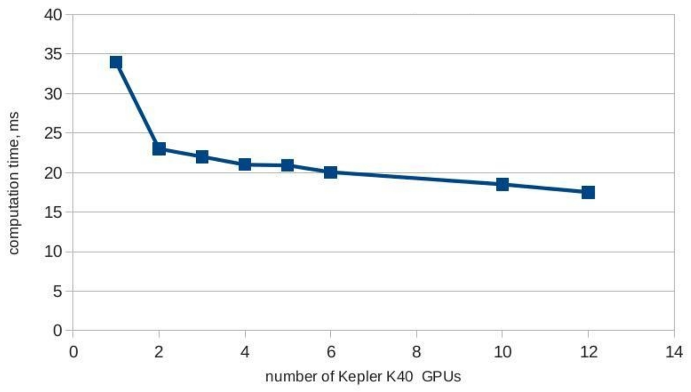


Figure 6. Computation time for the cluster with Kepler K40 GPUs. Here the size of the solved problem is constant and the workload of a single GPU (the number of particles per one GPU) is decreased with the increase of the number of GPUs

Also the speedup was measured for two clusters equipped with GPUs: first cluster with Tesla M2090 and the second with Kepler K40. A single timestep is computed in 34 ms with one Kepler K40 and 17.5 ms with 12 Kepler K40 GPUs (same grid size, model particles are distributed between GPUs), the speedup is shown in fig. 6. Parallel efficiency is over 90 %.

2.2.1. Efficiency for large number of GPUs

The behaviour of the code with large number of GPUs is of special interest. Due to this reason the special perfomance tests were conducted with the hybrid part of the Lomonosov supercomputer. First, the speedup was measured fig. 7. The speedup is relatively small because of the small number of model particles, still one can see that the plot has not reached saturation and is growing. It means that the problems of bigger size will be accelerated more effectively, with higher speedup. At the Lomonosov supercomputer with one Tesla GPU a single timestep

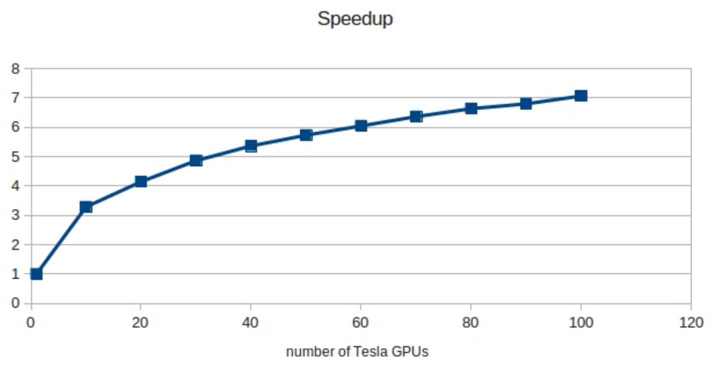


Figure 7. Speedup measured with the Lomonosov supercomputer.Here the size of the solved problem is constant and the workload of a single GPU (the number of particles per one GPU) is decreased with the increase of the number of GPUs

is computed (without communications) in 109 ms.

Now let us turn to the efficiency for the large number of GPUs (that is, up to 500), fig. 8. Here the number of model particles per one GPU (the workload of a GPU) is constant. It means that the size of the whole problem is growing with the number of GPUs. The sense of the test is to show what is the communication overhead for computations with large number of GPUs. Here, is there is no overhead at all and with 100 GPUs a 100 times bigger problem is solved in exactly the same time, the efficiency is 100 %. If the overhead is big, and with large number of GPUs most time is spent for communications then the efficiency will be low.

The efficiency in fig. 8 means that the time spent for a timestep does not change significantly with the number of GPUs. The preparation times such as initialization time, the time for copying the data to GPUs is not considered here. This is because in the real computation which lasts tens of thousands of timesteps all the preparation times will play no role, they will be very low in comparison with total time.

Let us remind that the workload of a single GPU (number of model particles) here is constant. The only communication between the GPUs is just the summation of currents: every GPU evaluates the current in the whole domain only with the particles residing on the GPU, after that one has to sum all the partial currents through all the GPUs. Thus only one MPI_Allreduce operation is performed per timestep and nothing more. Such a small number of MPI operations is because of co-design of the parallel algorithm.One must also notice that the average MPI_Allreduce time measured with Intel Trace Analyzer & Collector does not change significantly with the number of GPUs (at least up to 500): for 10 GPUs the MPI_Allreduce time was 29 ms and for 500 just 41 ms. And this is not because of co-design but because of the Lomonosov hardware and MPI installation.

This result is important not only as a performance test. Each GPU computed the motion of 320000 model particles, it means 160 million model particles as a whole. In such a way the code is really capable of doing physically meaningful simulations (as mentioned in the Introduction) with the Lomonosov supercomputer.

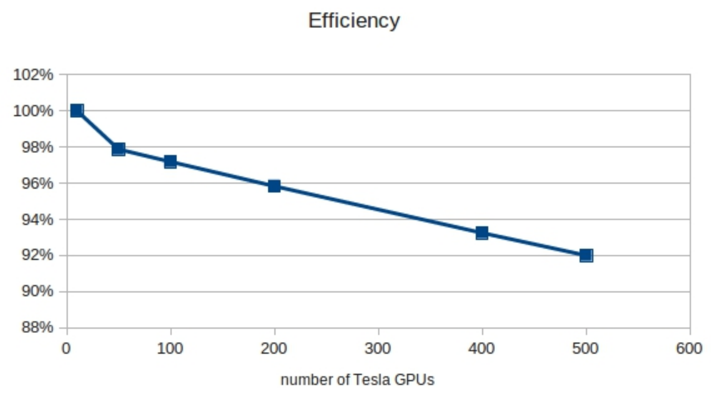


Figure 8. Parallelization efficiency measured with the Lomonosov supercomputer.

3. Simulation results

Finally let us consider the results of the two computational experiments in fig. 9 showing the simlation of the collision of two galaxies according to the described two-phase model. The details of the problem statement and simulation results can be found in [35].

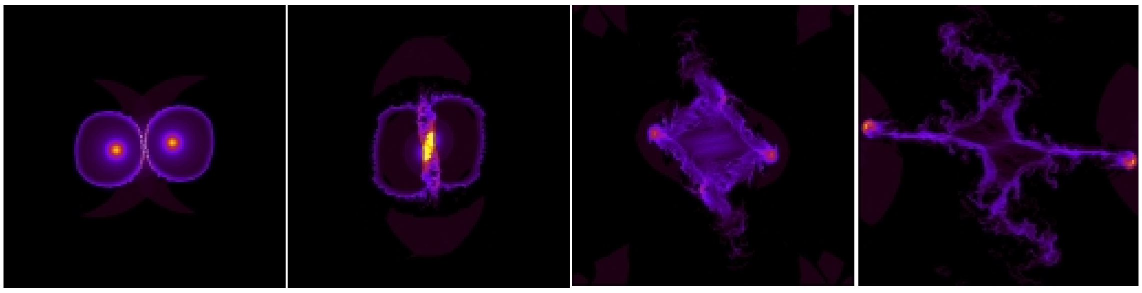


Figure 9. The runaway of galaxies after the central collision

4. Conclusion

The development of the efficient software for the hybrid supercomputers is a standalone complex scientific problem that requires the use of co-design at all stages from the physical statement of the problem to the software development tools. Within the numerical simulation context co-design is the design of physical and mathematical model, of the numerical method, of the parallel algorithm and of the implementation using the architecure of the hybrid supercomputer efficiently.

The use of co-design for the code for beam-plasma interaction simulation gives hope that the full 3D computations will be feasible. If one looks at the efficiency for 2048 processor elements (60 %) and the computation time for a single Kepler K40 GPU (34 ms), then, using about one thousand Kepler GPUs in a computational experiment one might obtain the necesary computational power for a full 3D simulation.

The co-design of the parallel numerical technology of the solution of the astrophysics problems resulted in 55 times speedup within a single GPU and 96% efficiency when using 60 GPUs, and also 27 times speedup in offload mode and 53 times speedup in native mode within a single Intel Xeon Phi and 94 % efficiency when using 32 MICs. Such an efficient use of the accelerators enable to conduct simulations with the grid size 1024^3 in acceptable time and to model "fine" effects in the collision of galaxies.

The present work was supported by the RFBR with grants 14-07-00241, 13-07-00589, 14-01-31088, 14-02-00294, 15-01-508, 15-31-20150 and 14-01-31199 and also by the Grant of the President of the Russian Federation for the support of young scientists number MK 6648.2015.9. We would like to thank Siberian Supercomputer Center department of ICMMG SB RAS for support of this work.

References

1. C.K. Birdsall, A.B Langdon, Plasma Physics via Computer Simulation. CRC Press, 01.10.2004 - 504 p.
2. A. Bret, L. Gremillet, and M.E. Dieckmann, Phys. Plasmas **17**, 120501 (2010).
3. I.V. Timofeev, K.V. Lotov, Phys. Plasmas **13**, 062312 (2006).
4. A. Burdakov, A. Arzhannikov, V. Astrelin, V. Batkin, A. Beklemishev, V. Burmasov, G.

Derevjankin, V. Ivanenko, I. Ivanov, M. Ivantsivskiy, I. Kandaurov, V. Konyukhov, I. Kotelnikov, K. Kuklin, S. Kuznetsov, K. Lotov, I. Timofeev, A. Makarov, M. Makarov, K. Mekler, S. Popov, S. Polosatkin, V. Postupaev, A. Rovenskikh, A. Shoshin, I. Shvab, S. Sinitsky, Yu. Suliaev, V. Stepanov, Yu. Trunyov, L. Vyacheslavov, V. Zhukov, and Eh. Zubairov, Fusion Sci. Technol. **55**(2T), 63 (2009).

5. M.K.A. Thumm, A.V. Arzhannikov, V.T. Astrelin, A.V. Burdakov, I.A. Ivanov, P.V. Kalinin, I.V. Kandaurov, V.V. Kurkuchekov, S.A. Kuznetsov, M.A. Makarov, K.I. Mekler, S.V. Polosatkin, S.A. Popov, V.V. Postupaev, A.F. Rovenskikh, S.L. Sinitsky, V.F. Sklyarov, V.D. Stepanov, Yu.A. Trunev, I.V. Timofeev, L.N. Vyacheslavov, Journal of Infrared, Millimeter, and Terahertz Waves, DOI: 10.1007/s10762-013-9969-3 (2013).
6. V.V. Postupaev, A.V. Burdakov, I.A. Ivanov, V.F. Sklyarov, A.V. Arzhannikov, D.Ye. Gavrilenko, I.V. Kandaurov, A.A. Kasatov, V.V. Kurkuchekov, K.I. Mekler, S.V. Polosatkin, S.S. Popov, A.F. Rovenskikh, A.V. Sudnikov, Yu.S. Sulyaev, Yu.A. Trunev and L.N. Vyacheslavov, Phys. Plasmas **20**, 092304 (2013).
7. I.V. Timofeev, Phys. Plasmas **19**, 044501 (2012).
8. A.V. Arzhannikov and I.V. Timofeev, Plasma Phys. Control. Fusion **54**, 105004 (2012).
9. V.A. Vshivkov, G. Lazareva, A. Snytnikov, I. Kulikov, A. Tutukov ”Hydrodynamical code for numerical simulation of the gas components of colliding galaxies” // The Astrophysical Journal Supplement Series. 2011. V. 194, 47. 12 pp.
10. Francesco Rossi, Pasquale Londrillo, Andrea Sgattoni, Stefano Sinigardi, Giorgio Turchetti. Towards robust algorithms for current deposition and dynamic load-balancing in a GPU particle in cell code. 2012. ADVANCED ACCELERATOR CONCEPTS: 15th Advanced Accelerator Concepts Workshop
11. I. Kulikov, I. Chernykh, A. Snytnikov, B. Glinskiy, A. Tutukov ”AstroPhi: A code for complex simulation of the dynamics of astrophysical objects using hybrid supercomputers” // Computer Physics Communications. 2015. V. 186. P. 71-80. DOI: 10.1016/j.cpc.2014.09.004
12. Yu.N. Grigoryev, V.A. Vshivkov, M.P. Fedoruk, Numerical particle-in-cell methods. Theory and applications. VSP, 2002.
13. Gingold, R.A., Monaghan, J.J., 1977. Smoothed particle hydrodynamics - Theory and application to non-spherical stars. MNRAS, 181, 375-389.
14. Lucy, L.B., 1977. A numerical approach to the testing of the fission hypothesis. ApJ. 82, 1013-1024
15. O’Shea, B., Bryan, G., Bordner, J., Norman, M., Abel, T., Harkness, R., Kritsuk, A., 2005. Adaptive Mesh Refinement - Theory and Applications. Lect. Notes Comput. Sci. Eng. 41, 341-350.
16. Pearcea, F.R., Couchman, H.M.P., 1997. Hydra: a parallel adaptive grid code. New Astronomy. 2, 411-427.
17. Wadsley, J.W., Stadel, J., Quinn, T., 2004. Gasoline: a flexible, parallel implementation of TreeSPH. New Astronomy. 9, 137-158.
18. Matthias, S., 1996. GRAPESPH: cosmological smoothed particle hydrodynamics simulations with the special-purpose hardware GRAPE. MNRAS. 278, 1005-1017.

19. Springel, V., 2005. The cosmological simulation code GADGET-2. MNRAS. 364, 1105-1134.
20. Ziegler, U., 2005. Self-gravitational adaptive mesh magnetohydrodynamics with the NIRVANA code. A&A. 435, 385-395.
21. Mignone, A., Plewa, T., Bodo, G., 2005. The Piecewise Parabolic Method for Multidimensional Relativistic Fluid Dynamics. ApJ. 160, 199-219.
22. Hayes, J., Norman, M., Fiedler, R. et al., 2006. Simulating Radiating and Magnetized Flows in Multiple Dimensions with ZEUS-MP. ApJS. 165, 188-228.
23. Teyssier, R., 2002. Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES. A&A. 385, 337-364.
24. Kravtsov, A., Klypin, A., Hoffman, Y., 2002. Constrained Simulations of the Real Universe. II. Observational Signatures of Intergalactic Gas in the Local Supercluster Region. ApJ. 571, 563-575.
25. Stone, J. et al., 2008. Athena: A New Code for Astrophysical MHD. ApJS. 178, 137-177.
26. Brandenburg, A., Dobler, W., 2002. Hydromagnetic turbulence in computer simulations. Comp. Phys. Comm. 147, 471-475.
27. Gonzalez, M., Audit, E., Huynh P., 2007. HERACLES: a three-dimensional radiation hydrodynamics code. A&A. 464, 429.
28. Krumholz, M.R., Klein, R.I., McKee, C.F., Bolstad, J., 2007. ApJ. 667, 626.
29. Mignone, A. et al., 2007. PLUTO: a Numerical Code for Computational Astrophysics. ApJS. 170, 228.
30. Almgren, A. et al., 2010. CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity. ApJ. 715, 1221.
31. Schive, H., Tsai, Y., Chiueh, T. 2010. GAMER: a GPU-accelerated Adaptive-Mesh-Refinement Code for Astrophysics. ApJ. 186, 457-484.
32. K.V.Lotov, I.V.Timofeev, E.A.Mesyats, A.V.Snytnikov, V.A.Vshivkov. Note on quantitatively correct simulations of the kinetic beam-plasma instability. http://arxiv.org/abs/1410.5617
33. N. Mitchell, E. Vorobyov, G. Hensler "Collisionless Stellar Hydrodynamics as an Efficient Alternative to N-body Methods" // Monthly Notices of the Royal Astronomical Society. 2013. Vol. 428, I. 3. P. 2674-2687.
34. I. Kulikov "PEGAS: Hydrodynamical code for numerical simulation of the gas components of interacting galaxies" // The Book Series of the Argentinian Association of Astronomy. 2013. Vol. 4. P. 91-95.
35. I. Kulikov "GPUPEGAS: A New GPU-accelerated Hydrodynamic Code for Numerical Simulations of Interacting Galaxies" // The Astrophysical Journal Supplement Series. 2014. V. 214, 12. 12 pp. Arxiv: http://arxiv.org/abs/1311.0861