Parallelization of iterative methods to solve sparse linear systems using task based runtime systems on multi and many-core architectures: application to Multi-Level Domain Decomposition methods

Adrien Roussel

Tuesday 6th February, 2018

Preamble

- INRIA
  - Public research institute
- Area of research
  - Applied mathematics
  - Computer Science
- MOAIS team (Grenoble)
- AVALON team (Lyon)
  - High performance computing
    - Scheduling

- IFP Energies Nouvelles
  - EPIC
- Area of research
  - Renewable energy
  - Transport
  - Oil & Gas industry
- Computer science division
  - Basin modeling
  - Numerical simulation in reservoir engineering

Outline

1 Context and Introduction

2 Sparse Linear Algebra API

3 Performance portability on Many-core systems

4 Domain Decomposition Methods

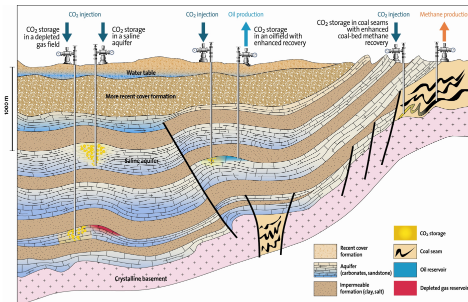
5 Conclusion

Numerical simulations in reservoir engineering

- Strategic stakes
  - Numerical simulator performances
- Need to replay simulations many times
  - To explore new scenarios
- Main issue
  - Intensive computational effort

- Performance improvements
  - Simulation's precision
  - Increasing the number of simulations

Numerical simulations in reservoir engineering



- Porous Media Flow models
- Partial Differential Equations
  - Finite Volume scheme
    - Non-linear system solved with Newton method
- Linear Solvers
  - up to 80% of computing load

Issues on linear solver performances

- Challenges
  - Linear solvers
    - Complex methods
  - Linear systems
    - Large
    - Sparse
    - Unstructured
- Standard approaches
  - BiCGStab
  - GMRES
- Preconditioned systems
  - Improving convergence rates
  - Examples:
    - Polynomial
    - Incomplete LU Factorization (ILU)
    - Algebraic Multi-Grid methods (AMG)

Parallel architectures

- Hardware evolution
  - Hierarchical memory
    - Interconnected memory banks
    - Cache hierarchy
  - Increasing number of cores
    - Example: Many-core processors
  - Heterogeneity
    - GPU
    - FPGA

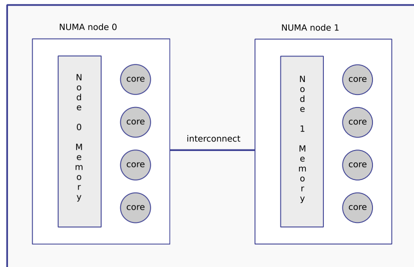


Figure: NUMA design

DDML as a promising method to scale on emerging architectures

- Standard methods
  - ILU(0)
    - Not robust on reservoir case
    - Difficulties to scale on large machine
  - AMG
    - Complex parallelization
- Multi-Level Domain Decomposition methods
  - Numerical properties
    - Robust
    - Extensible
  - Parallelization
    - Naturally parallel
    - Fine grain size

How to program parallel architectures ?

- Top-down classification
  - Network
    - MPI
  - Multi-core
    - OpenMP, Cilk, Intel TBB, StarPU, X-Kaapi, OmpSs, HARTS. . .
  - Accelerators
    - OpenCL, CUDA, OpenACC, OpenMP (since 4.0), OmpSs
  - SIMD unit
    - Compiler, AVX, OpenMP (since 4.0)

- Programming parallel computers is challenging !
  - No consensus on standard way to program parallel computers
    - Trend: Task based programming model

Task programming model

- Advantages
  - Fine control on parallelism
- Task description
  - Piece of work
  - Data dependencies
    - Input
    - Output
- Organization
  - Direct Acyclic Graph (DAG)
- Implementation
  - Runtime System

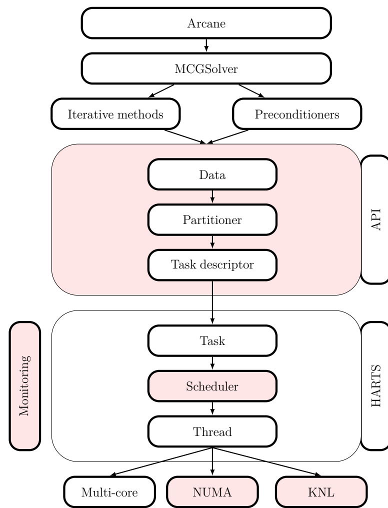
Task-based Runtime Systems

Features	OpenMP	OmpSs	X-Kaapi	HARTS	StarPU
Pragma directive	Yes	Yes	Yes	No	No
Scheduling policy	Impl. dep.	Various	Work Stealing	Central. queue	Various
Dataflow model	Yes	Yes	Yes	No	Yes
Persistent structures	No	No	No	Yes	No
Data locality	No	Yes	Yes	No	No
GPU	Yes	Yes	Yes	Yes	Yes

Problematic

- Writing complex parallel sparse linear algebra problems...
  - Numericians are no longer able to handle hardware complexity
- ... While guaranteeing good performances for users
  - Able to survive to hardware evolution
    - Performances portability
    - Extensibility
- Methodology
  - Construction of a sparse linear algebra interface
  - Enable performances portability to take into account emerging architectures
  - Performance study: Multi-level Domain Decompositon preconditioner

Overview



Contents

1 Context and Introduction

2 Sparse Linear Algebra API

3 Performance portability on Many-core systems

4 Domain Decomposition Methods

5 Conclusion

Outline

1 Context and Introduction

2 Sparse Linear Algebra API

3 Performance portability on Many-core systems

4 Domain Decomposition Methods

5 Conclusion

Introduction

- Motivations
  - **Programming**
    - Provide users a set of functions to write standard iterative methods
    - Implicit parallelism
  - **Efficient execution**
- Widespread kernels in iterative methods
  - BLAS Level 1 & 2
  - Specific sparse kernels
    - Sparse Matrix Vector product (SpMV)
    - Preconditioners

Algorithm 1: BiCGStab Algorithm

```
Matrix A;
Vector b, pp, p, r, v;
Preconditioner P;
Scalar a;
do
    pp = inv(P).p;
    v = A.pp ;
    r += v;
    a = dot(p,r);
    if(a==0) break;
    ...;
while (|r| < tol * |b|);
```

Abstract Linear Algebra API

- Goals
  - **Sequential semantic** to write iterative methods
  - **Implicit** parallelism
- Key concepts
  - Capture **Iterative pattern**
- Integrates
  - Executive layer
    - Tasks description
    - Tasks and dependencies are sent to **runtime system** for execution
  - **Data manager**
    - Allocation
    - Partitioning
- Various implementations
  - OpenMP 4.0
  - OmpSs
  - X-Kaapi
  - HARTS

API's functions

```
class AlgebraKernel
{
    // allocates 'size' bits and returns it in 'v' ;
    void allocate(size_t size, Vector& v) ;
    // Out: y = op() ;
    void assign(LambdaT op, Vector& y) ;
    // Out: y = x ;
    void copy(Vector const& x, Vector& y) ;
    // Out: y = a * y
    void scal(Value const &a, Vector& y) ;
    // Out: y += a * x ;
    void axpy(Value const &a, Vector const& x, Vector& y) ;
    // Out: a = ( x . y ) ;
    void dot(Vector const& x, Vector const& y, Value& a) ;
    // Out: y = A * x ;
    void mult(Matrix const& A, Vector const& x, Vector& y) ;
    // Out: preconditioner 'P' applied to 'x' and stored in 'y' ;
    void exec(Precond const& P, Vector const& x, Vector& y) ;
    // Break if value is null ;
    void assertNull(double value) ;
    // Performs the Sequence objects referred by "id" ;
    void process(SequenceType id) ;
} ;
```

Generation of parallelism

- Task generation
  - Data partitioning
    - Vector
    - Matrix
- Computed from
  - Graph partitioning techniques
    - Adjacency graph of the coefficient matrix
- Considerations
  - Load balancing
  - Communication reduction
    - Limiting connections between subgraphs (*i.e.* dependencies)
- Reordering
  - Contiguous memory accesses
- Graph partitioners
  - Row partitioning
  - Metis partitioning
    - Heuristics to ensure load balancing & to minimize dependencies

Execution Layer

- Parallel processing
  - DAG execution
- Be able to conserve or to build on-the-fly the DAG
  1. Storing execution DAG
     - HARTS
  2. Storing all the necessary information to build the DAG with Task Descriptor objects
     - OpenMP 4.0
     - OmpSs
     - X-Kaapi
- Storing DAG construction information
  - Sequence object
    - List of operations
    - Capturing iterative pattern

Task creation from Partitioning techniques

1. Operation $x \leftarrow \alpha.x$
   - Input: α
     - Scalar
   - Input/Output: x
     - Vector

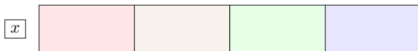
```
alg.scal(alpha, x);
```

x

Task creation from Partitioning techniques

1. Operation $x \leftarrow \alpha.x$
   - Input: α
     - Scalar
   - Input/Output: x
     - Vector
2. Partitioned x

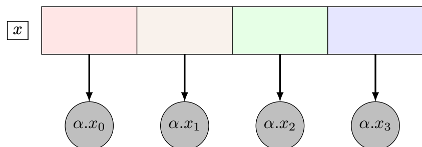
```
alg.scal(alpha, x);
```



Task creation from Partitioning techniques

1. Operation $x \leftarrow \alpha.x$
   - Input: α
     - Scalar
   - Input/Output: x
     - Vector
2. Partitioned x
3. Creation of the task descriptors

```
alg.scal(alpha, x);
```



Task creation from Partitioning techniques

1. Operation $x \leftarrow \alpha.x$
   - Input: α
     - Scalar
   - Input/Output: x
     - Vector
2. Partitioned x
3. Creation of the task descriptors
4. Instanciation in the runtime system

```
Sequence.process();
```

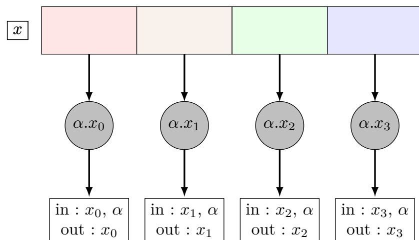
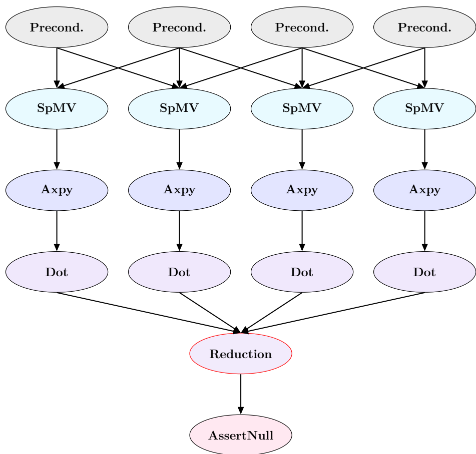


Illustration – BiCGStab method

```
AlgebraKernelType alg;
Matrix A; Vector p,pp,r,v;
double alpha;
SequenceType seq =
  alg.newSequence();
alg.exec(precond,p,pp,seq) ;
alg.mult(A,pp,v,seq);
alg.axpy(1.,r,v,seq);
alg.dot(p,r,alpha,seq);
alg.assertNull(alpha,seq);
while(!iter.stop())
{
  alg.process(seq);
}
```

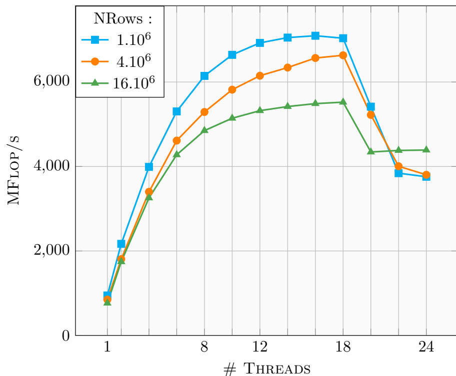


Evaluation of the API

- Laplacian systems
  - Origin
    - Discretization of a 2D Laplace problem
  - Interest
    - Variable size
- SPE10
  - Origin
    - Realistic oil reservoir simulation
  - Interest
    - Ill-conditioned system

First evaluation of the API

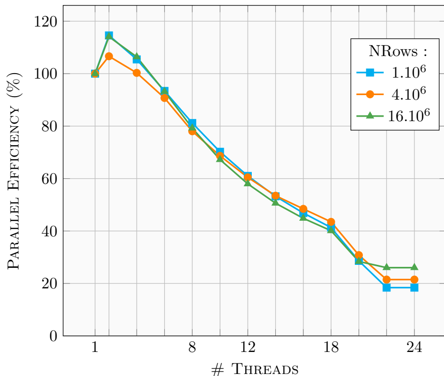
- Evaluation
  - BiCGStab
- Input matrix
  - Laplacian system
- Machine
  - 1 × 24 cores Broadwell processor
- Measurements
  - MFlops



- Observation
  - Adapted programming model

First evaluation of the API

- Evaluation
  - BiCGStab
- Input matrix
  - Laplacian system
- Machine
  - 1 $\times$ 24 cores
    Broadwell processor
- Parallel Efficiency
  - $\text{Eff}(p) = \frac{T_1}{p\ T_p}$
- Observation
  - Bad efficiency

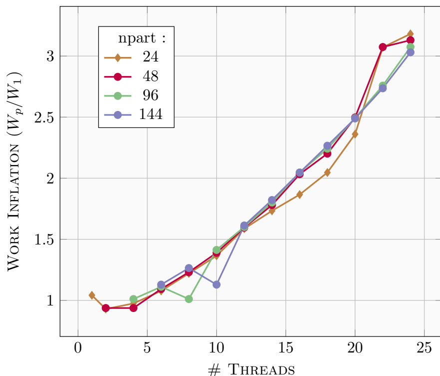


Why and how to increase performances ?

- Need to analyze the execution
  - Monitoring tools and performance counters
    - At runtime level, in HARTS
- Measurements
  - Tasks duration
  - Placement
  - Threads activity
  - Output
    - Gantt chart
    - Critical path

Work inflation on a BiCGStab method

- Input case
  - Laplacian matrix
- Machine
  - 24 cores Broadwell processor
- Measurement
  - Work $= \sum \text{Duration}(T_i)$
- Work increases in function of the number of threads in use
  - Previously studied in (Olivier et al., 2012)
- Reasons
  - Limited bandwidth
    - More important on NUMA due to remote memory accesses



Data locality management for NUMA architectures

- How to limit work inflation ?
  - Reducing data communication across NUMA nodes
- Data locality management in runtime systems
  - OpenMP, HARTS
    - Nothing
  - OmpSs (Ayguadé et al., 2010)
    - `Socket` scheduler
    - Initialization tasks
  - X-Kaapi, libKomp (Virouleau, Broquedis, et al., 2016; Virouleau, Roussel, et al., 2016)
    - Work-stealing based on data dependencies locality
    - OpenMP extensions through affinity clause

Managing data locality in HARTS

- 2 steps approaches

1. Data distribution across NUMA nodes
   - `numactl`
     - Insufficient control
   - Parallel initialization
     - Initialization tasks
     - First-touch NUMA policy

2. Work Stealing heuristic to take care of data locality

An heuristic designed to favor data locality

- Parallel initialization tasks
  - Static scheduling
    - Association between a thread and a range of partition ids
  - Assumption on NUMA first-touch policy to distribute data
    - All data relative to the same partition identifier are in the same NUMA memory bank
- Ready tasks
  - Inserted in the queue related to task's partition id
- Per-worker task queue
  - Only filled with tasks operating close data
- Heuristic
  - Steal a task which operates on close data
    - NUMA distance between threads
  - Pre-computed priority list of potential victims (*i.e.* threads)
    - One per thread

Experimental protocol

- 2×14 cores Broadwell processor
  - 2.40 GHz
- Configuration
  - Cluster-on-die
    - A processor is viewed as two NUMA nodes
- Memory
  - 128 Go

Experiment without and with heuristic to enhance data locality on a BiCGStab method ($p = 28$ cores)

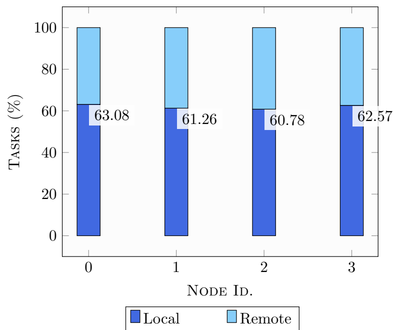


Figure: Centralized queue

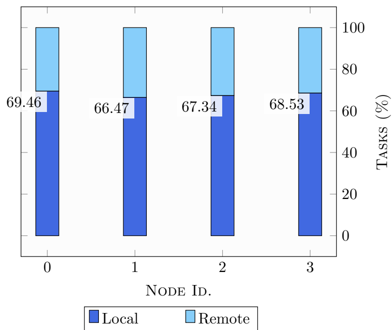
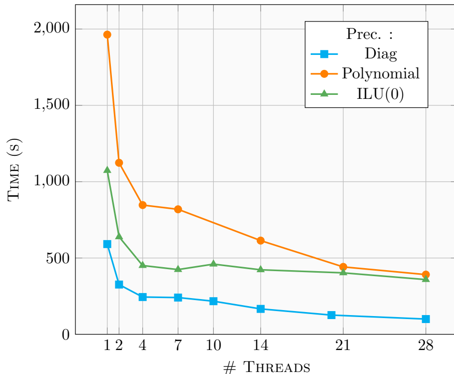


Figure: Distributed queue

- Conclusion
  - Slight improvement of data locality task placement

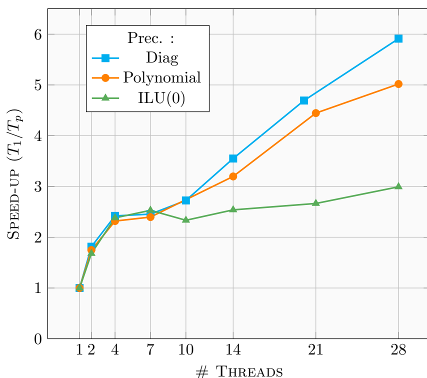
API Evaluation on a preconditioned BiCGStab method with HARTS

- Input case
  - Laplacian matrix
    - $N_{rows} = 16.10^6$
- Measurement
  - Elapsed time
- Conclusion
  - Time decrease in function of threads



API Evaluation on a preconditioned BiCGStab method with HARTS

- Input case
  - Laplacian matrix
    - $N_{rows} = 16.10^6$
- Measurement
  - Speed-up ratio (T_1/T_p)
- Conclusion
  - ILU(0) fails
    - Insufficient parallelism



Conclusion

- ▷ Abstract linear algebra API
  - ◦ Balance between
    - Expressiveness
    - Performances
  - ◦ Keep it simple
    - Sequential semantic
    - Implicit parallelism
- ▷ Performances
  - ◦ Scheduling could be improved to increase performances
- ▷ How to take care of variable hardware evolution ?

Outline

1 Context and Introduction

2 Sparse Linear Algebra API

3 Performance portability on Many-core systems

4 Domain Decomposition Methods

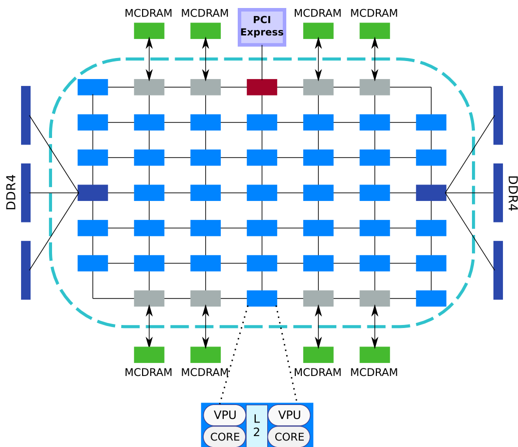
5 Conclusion

Hardware evolution

- HARTS and the API originally designed for
  - Multi-core architectures
- Hardware trends in parallel computing
  - GPU computing
  - FPGA
  - Many-core processors
- Main issue
  - Performance portability
    - Guaranteeing application's performances regardless the architecture

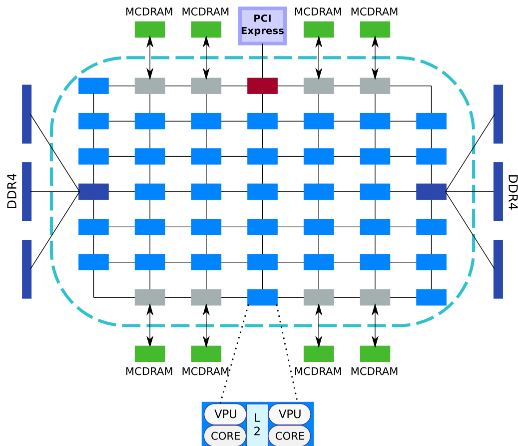
Knights Landing architecture

- Many-core processor
  - up 72 cores
  - 36 bi-core processors with a 2D mesh interconnect
    - 2 $\times$ VPU
    - Shared L2 cache memory
  - 16 Go of MCDRAM, High bandwidth memory
- Complex configuration
  - Cluster Mode
    - All-to-all
    - Quadrant Hemisphere
    - Sub-Numa cluster
  - Memory mode
    - Flat
    - Cache
    - Hybrid



Programming challenges

1. Number of cores
   - from 64 to 72
     - Fine grain parallelism
2. Vectorization
   - Core simplification
     - Lower frequency ($\simeq$ 1.3 GHz)
     - AVX-512
3. Memory management
   - DRAM
     - Huger memory capacity
   - MCDRAM
     - High bandwidth
     - but limited size



Are all kernels vectorized ?

- Widespread kernels in iterative methods
  - Blas Level 1 & 2
    - Easily vectorized
  - Specific sparse kernels
    - Need to change the sparse matrix representation
- Data representation impact
  - Compressed Sparse Rows (CSR)
    - Eigen (Guennebaud, Jacob, et al., 2010)
    - Intel Sparse MKL (Intel, 2017)
    - PETSc (Balay et al., 1997)
  - Difficult to vectorize
- Vector-friendly format
  - Ellpack format (Grimes et al., 1980)
    - May generate too much padding
  - Sell-C-σ (Kreutzer et al., 2013)
    - Limiting memory footprint

Data structures management

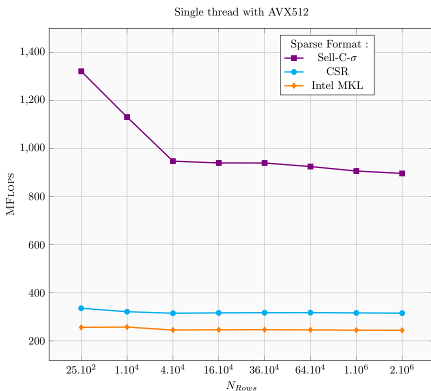
- Issues
  - Managing several structures for one representation
  - Kernel selection to enable specific implementation

Different possibilities

- Multi-versioning kernels in Runtime systems
  - Examples
    - StarPU (codelet) (Augonnet and Namyst, 2008)
    - X-Kaapi (Body) (Gautier et al., 2013)
- Dynamic dispatch to enable auto-tuning
  - Detecting hardware features to select appropriate kernel implementation
  - Example
    - Intel MKL (Intel, 2017)
- Compile time selection
  - No need to maintain several structures at runtime
  - Adapted to application context
  - Limiting factor at middle term (*e.g.* heterogeneous architectures)

Single thread execution – AVX512

- Input case
  - Laplacian matrix
  - Different sizes
- Measurement
  - Operations performed per second
- Configuration
  - 1 thread
- Conclusion
  - $3\times$ more performances with a good vectorization



Multi-threaded comparison

- Input case
  - Laplacian matrix
  - Different sizes
- Measurement
  - Operations performed per second
- Configuration
  - Multi-threaded execution

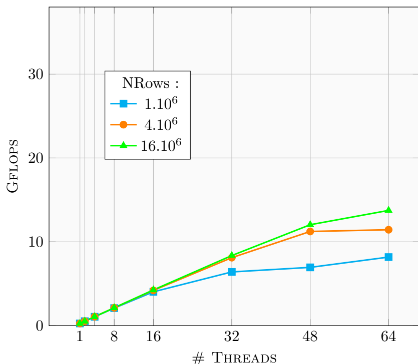


Figure: CSR

Multi-threaded comparison

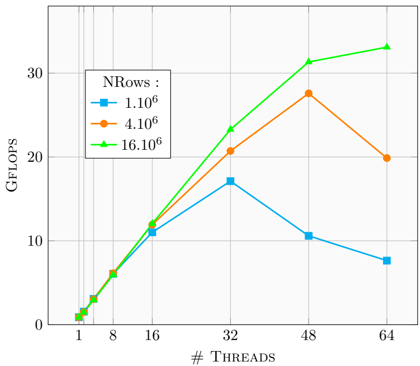


Figure: Sell-C-σ

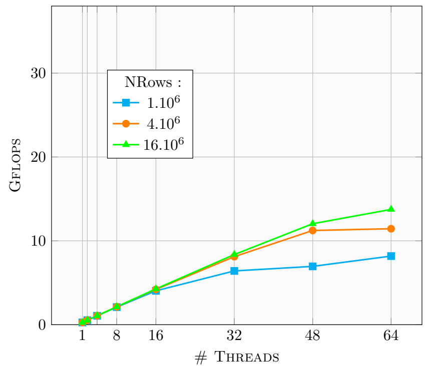


Figure: CSR

Managing memory allocation

- How to benefit from MCDRAM memory ?
  - Cache mode
    - Management at hardware level
  - Flat mode
    - Explicit management by developers
    - *Memkind* library: allocating memory in MCDRAM
- Memory management
  - Runtime level
    - Distributed Shared Memory (DSM)
    - Example: StarPU (Augonnet and Namyst, 2008)
  - Management in the API while using `Allocator` concept
    - Compile time selection
    - All data fit in MCDRAM bank

Impacts of MCDRAM on performances

- Input case
  - Laplacian matrix
  - Different sizes
- Benchmark
  - SpMV
- Measurement
  - Operations performed per second
- Configuration
  - 64 threads
  - Flat memory mode
- Conclusion
  - MCDRAM reaches up to 7× more performances

Cluster modes impact on a BiCGStab method with HARTS

- Input case
  - Laplacian matrix
    - $N_{rows} = 4.10^6$
- Configuration
  - Flat memory mode
    - MCDRAM memory allocation
- Measurement
  - Operations performed per second
- Conclusion
  - No impact

Cluste Mode :
All-to-All
Quadrant
SNC4

GFlops
Threads

Conclusion

- Abstract Linear algebra API
  - Extensibility
    - Adaptation to new architectures
- Performances portability on KNL
  - Vectorization
    - Data structure management
    - Kernel selection
  - Memory management

- Perspectives
  - Dynamic dispatch
    - Automatic detection of architectures
    - No need of user knowledge

Outline

1 Context and Introduction

2 Sparse Linear Algebra API

3 Performance portability on Many-core systems

4 Domain Decomposition Methods

5 Conclusion

Motivation

- Main interests
  - Numerical performances
  - Scalability
    - Fine grain parallelism (Jolivet et al., 2013)
- 2-level Additive Schwarz Method
  - Solving problem from subdomain solutions
    - Local solvers
  - Coarse Operator
    - Couple all the subdomains
    - Based on the computation of an eigenvalue problem
- Portability above the API
  - Task decomposition

Task-based parallelization – Direct Acyclic Graph

- Computational intensive tasks
  - Coarse Op
  - Local Solv.
- Parameters and Influence
  - Coarse Op
    - Eigen vectors numbers
    - Partitioning
  - Local Solvers
    - System size
    - Partitioning

DDML tasks distribution in percentage ($p = 28$ cores) on SPE10 benchmark

Figure: $N_{part} = 48$

Figure: $N_{part} = 448$

- Conclusion
  - Compromise between CoarseOp and LocalSolver

Experimental compromises between LocalSolv and Coarse Op

- ▷ Input case
  - SPE10 coming from oil reservoir simulation
- ▷ Measurement
  - Task granularity
- ▷ Objective
  - Optimal sizes of the two main tasks of DDML

SPE10
$\cdot 10^5$
$\cdot 10^4$
Local Solv
Coarse Op.
N_{ev} :
2
5
10
\# partitions

DDML evaluation with HARTS on multi-core system

- Input case
  - SPE10 matrix
- Measurement
  - Elapsed time on p cores
- Conclusion
  - DDML is challenging face to AMG

DDML evaluation with HARTS on KNL many-core processor

- Input case
  - SPE10 matrix
- Measurement
  - Elapsed time on p cores
- Objective
  - Performance analysis

ILU0
DDML
AMG
TIME (S)
150
100
50
0
1
8
16
32
48
64
THREADS

Conclusion & perspectives

- Good alternative to AMG
  - Fine grain parallelism
  - Convergence rates
    - Oil reservoir case
- Remaining optimizations
  - Many-core processors
    - Direct solvers vectorization
- Improving convergence rates
  - Updating coarse operator definition

Outline

1 Context and Introduction

2 Sparse Linear Algebra API

3 Performance portability on Many-core systems

4 Domain Decomposition Methods

5 Conclusion

Conclusion

- An abstract Linear Algebra API
  - Sequential semantic and implicit parallelism
    - Partitioning
    - Task programming model
    - Runtime system execution
  - Extensions in HARTS
    - Monitoring
    - Data locality aware scheduler
- Extensions to support KNL many-core processors performance portability
  - Vectorization
  - MCDRAM management
- Application to Multi-Level Domain Decomposition methods
  - Encouraging results on multi and many-core systems

Perspectives

- Programming model
  - Code evolution face to hardware variations
    - KOALA ANR proposal
- Work inflation limitation
  - Limited bandwidth
    - Limits of parallelism

Thanks for your attention

Matrices Laplaciennes

Mesh size	N_R	N_{NZ}	Memory Footprint
1000 x 1000	1.10^6	4996000	488 Mb
2000 x 2000	4.10^6	19992000	1953 Mb
3500 x 3500	12.10^6	61236000	5980 Mb
4000 x 4000	16.10^6	79984000	7811 Mb

Table: Test cases overview

API Evaluation on a preconditioned BiCGStab method with HARTS

- Input case
  - Laplacian matrix
- Measurement
  - MFlops

API Evaluation on a preconditioned BiCGStab method with HARTS

- Input case
  - Laplacian matrix
- Measurement
  - Computational Efficiency