

Comparing Programmability and Scalability of Multicore Parallelization Paradigms with C++

Christian Terboven, Christopher Schleiden, Dieter an Mey
{terboven, schleiden, anmey}@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University, Germany

Agenda

- Motivation and Computational Task
- Overview of Parallelization Paradigms
- Programmability Evaluation
- Performance and Scalability Evaluation
- Conclusion and Future Work

2

Center for

Computing and

Communication

Motivation

Paradigms

Programmability

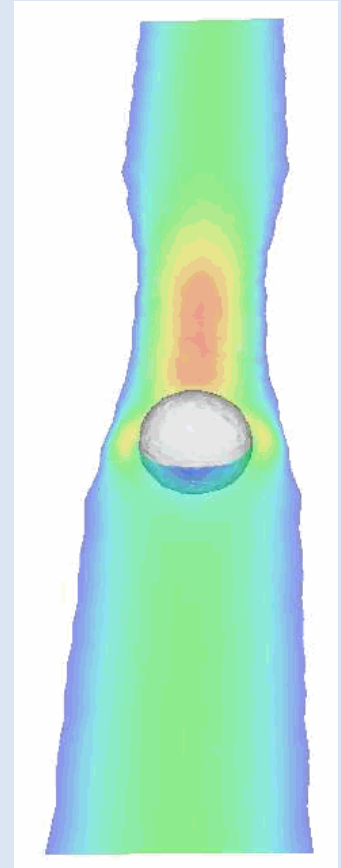
Scalability

Conclusion

DROPS: A Navier-Stokes Solver in C++

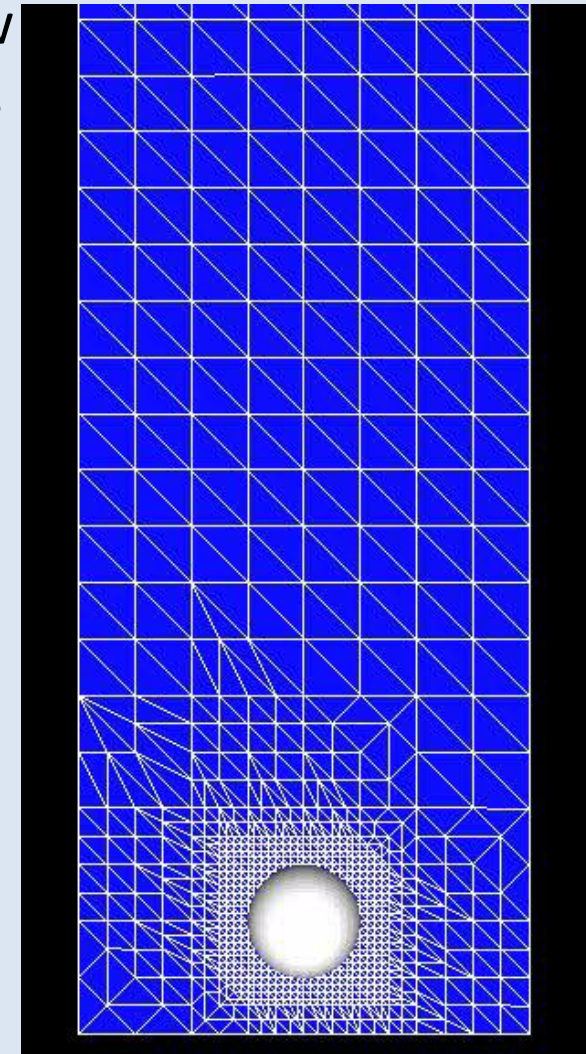
- Numerical Simulation of two-phase flow
- Modeled by instationary and non-linear Navier-Stokes equation
- Level Set function is used to describe the interface between the two phases
- Written in C++: is object-oriented, uses nested templates, uses STL types, uses compile-time polymorphism, ...
- (Adaptive) Tetrahedral Grid Hierarchy
- Finite Element Method (FEM)

Example:
Silicon oil drop in
 D_2O (fluid/fluid)



DROPS: A Navier-Stokes Solver in C++

- Numerical Simulation of two-phase flow
- Modeled by instationary and non-linear Navier-Stokes equation
- Level Set function is used to describe the interface between the two phases
- Written in C++: is object-oriented, uses nested templates, uses STL types, uses compile-time polymorphism, ...
- (Adaptive) Tetrahedral Grid Hierarchy
- Finite Element Method (FEM)

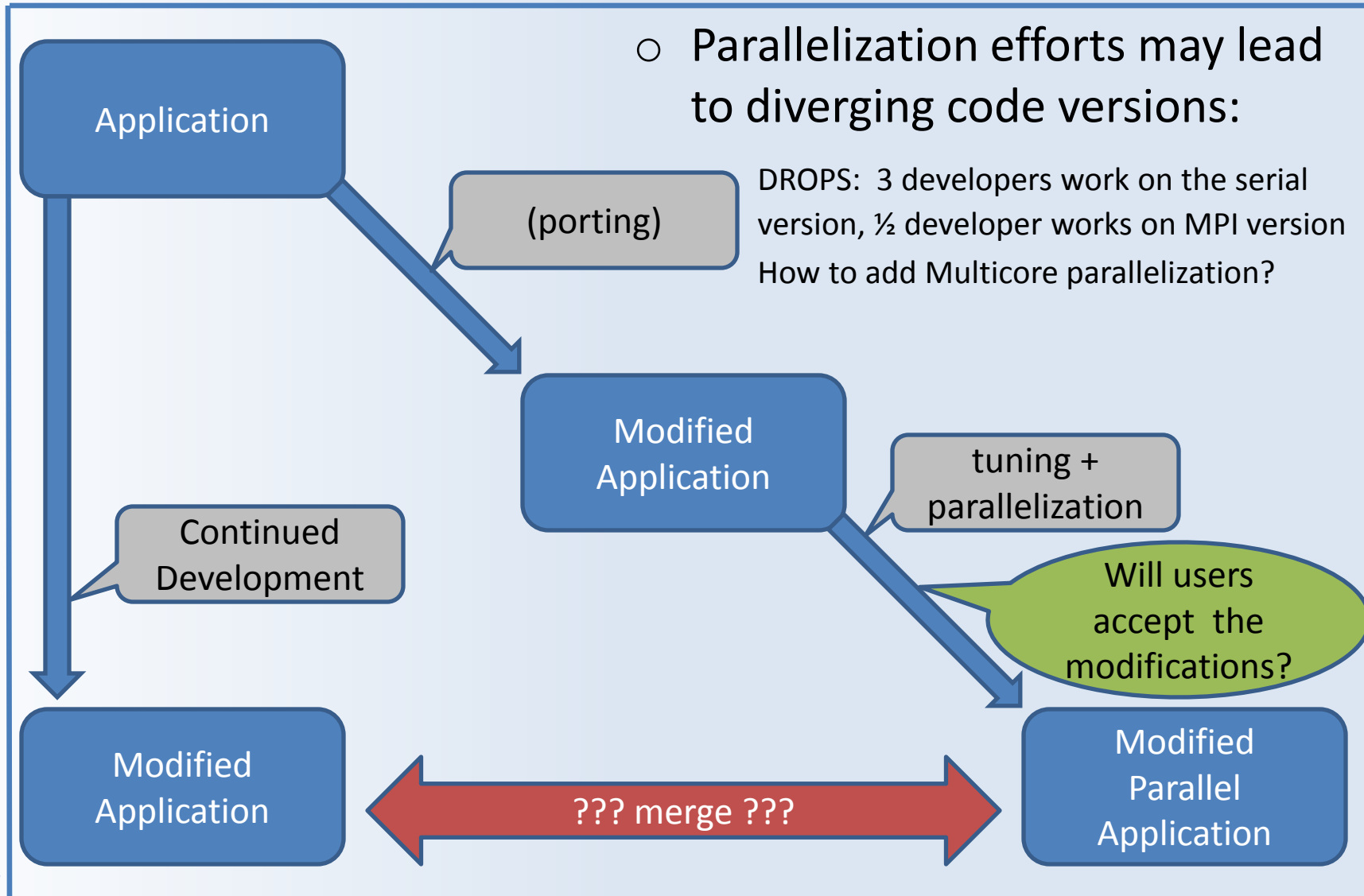


4

Parallelization may lead to Dead Ends

- Parallelization efforts may lead to diverging code versions:

DROPS: 3 developers work on the serial version, ½ developer works on MPI version
 How to add Multicore parallelization?



5

DROPS: Iteration Loop of CG-type solver

- Our targets:
 - Efficient parallelization not hindering development
 - Maintain the object-oriented programming style

```
PCG(const MatrixCL& A, VectorCL& x, const VectorCL& b,  
    const PreCon& M, int& max_iter, double& tol)  
{  
    VectorCL q(n), p(n), r(n);  
    [...]  
    for (int i=1; i<=max_iter; ++i) {  
        [...]  
        q = A * p;  
        double alpha = rho / (p*q);  
        x += alpha * p;  
        r -= alpha * q;  
        [...]  
    }  
}
```

Evaluation of:

- OpenMP (2.5 and 3.0)
- Threading Building Blocks
- Native Threading

6

enter for

Computing and

Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Agenda

- Motivation and Computational Task
- Overview of Parallelization Paradigms
- Programmability Evaluation
- Performance and Scalability Evaluation
- Conclusion and Future Work

7

Multicore Parallelization Paradigms

- OpenMP: fork-join parallelism
 - Fortran, C and C++
 - Parallel Region: pragma + structured block
 - Worksharing and Task-based parallelization
- (Intel) Threading Building Blocks: library-based parallelism
 - C++ template library, global Task scheduler
 - Worksharing-like expressions and parallel STL-type algorithms
 - Everything is a Task concept (implicit or explicit Tasks)
- Posix-Threads / Win32-Threads: native threading
 - API to create and manage OS-level threads
 - Worksharing has to be implemented manually
- Other: Hardly portable / supportable

8

Center for

Computing and

Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Agenda

- Motivation and Computational Task
- Overview of Parallelization Paradigms
- **Programmability Evaluation**
- Performance and Scalability Evaluation
- Conclusion and Future Work

9

Center for

Computing and
Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Parallelization: Naive Approach w/ OpenMP

```
PCG(const MatrixCL& A, VectorCL& x, const VectorCL& b,  
    const PreCon& M, int& max_iter,  
    double& tol)  
{  
    VectorCL p(n), z(n), q(n), r(n);  
    [...]  
    for (int i=1; i<=max_iter; ++i)  
        [...]  
        q = A * p;  
        double alpha = rho / (p*q);  
        x += alpha * p;  
        r -= alpha * q;  
        [...]
```

Option 1: Replace operator calls

```
y_Ax_par(&q.raw()[0],  
        A.num_rows(), A.raw_val(),  
        A.raw_row(), A.raw_col(),  
        Addr(p.raw()));
```

Option 2: Place parallelization inside operator calls

○ Problems of both options:

- Code Changes
- Parallelization not applicable to complex expressions
- Parallelization may introduce additional overhead

Parallelization: New Approach via Library

- Extend existing abstractions to introduce parallelism!

- For the DROPS code, just replace

```
typedef VectorBaseCL<double>      VectorCL;  
typedef SparseMatBaseCL<double> MatrixCL;
```

- with

```
typedef laperf::vector<double,  
                      OpenMPInternalParallelization>  
                      VectorCL;  
typedef laperf::matrix_crs<double> MatrixCL;
```

element data type

parallelization
type

- Specification of parallelization type by the user allows to
 1. Experiment with several parallelization strategies
 2. Mix and match parallel and sequential code

11

enter for

Computing and

Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Implementation History

- V1: Using C++ operator overloading and giant switch statements to differentiate between different parallelization paradigms → difficult to maintain and extend, performance problems because of temporary copies
- V2: Using inheritance to improve maintainability → still problems with temporary copies
- Current: V3: No inheritance; using C++ Template Expressions to avoid unnecessary temporary copies
- Future: V4: Policy-based design („Mix-In“); using C++0x lambda functions and move semantics

12

Problem: Temporaries

```
laperf::vector<double> x(dim), a(dim), b(dim);  
x = (a * 2.0) + b;
```

Users'
Code

ideal code for this vector operation:

```
for( int i = 0; i < dim; ++i )  
    x[i] = a[i] * 2.0 + b[i];
```

but in C++ it translates to:

```
laperf::vector<double> _t1 = operator*(a, 2.0);  
laperf::vector<double> _t2 = operator+(_t1, b);  
x.operator=(_t2);
```

- ➔ two temporary vector copies and unnecessary overhead
- ➔ impossible to implement efficient parallelization
- ➔ bad placement of temporaries on cc-NUMA architectures

13

enter for

Computing and

Communication

Motivation

Paradigms

Programmability

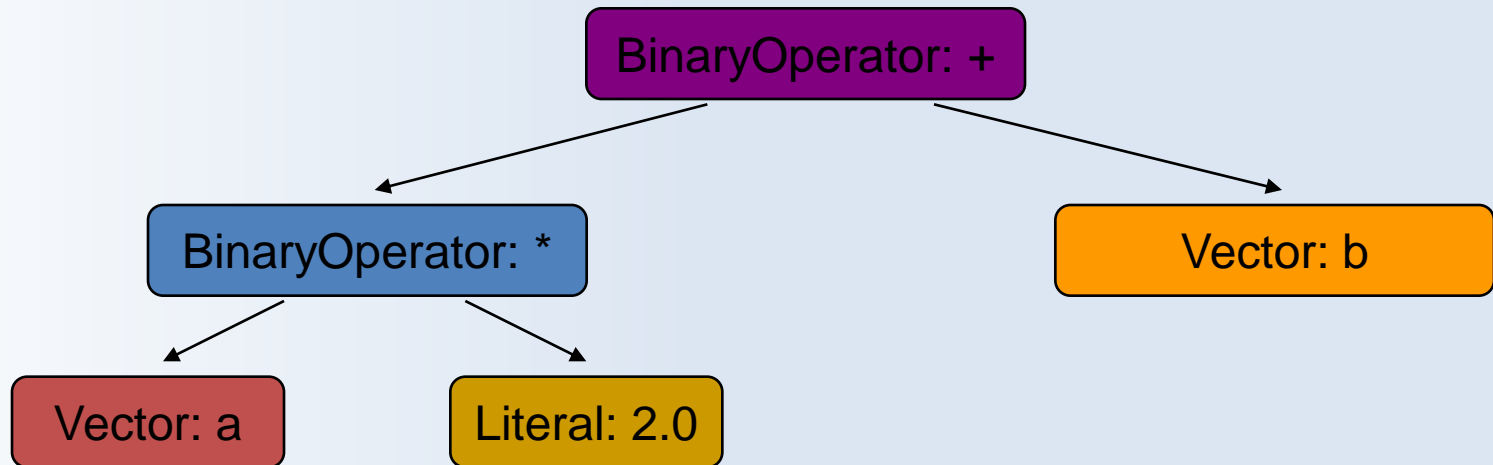
Scalability

Conclusion

Operator evaluation with Template Expressions

During compile time the compiler builds a template expression resembling the syntax tree:

```
x = (a * 2.0) + b;
```



```
LB<OpAdd, LB<OpMul, vector, double>, vector>  
expr ( LB<OpAdd, LB<OpMul, vector, double>, vector> (  
    LB<OpMul, vector, double>(a, 2.0), b  
    )  
);
```

Achieves 95-99.5% efficiency of hand-coded C while still having all benefits of C++'s object orientation

Parallelization of Template Expressions

```
LB<OpAdd, LB<OpMul, vector, double>, vector>  
expr( LB<OpAdd, LB<OpMul, vector, double>, vector>(  
    LB<OpMul, vector, double>(a, 2.0), b  
    )  
);
```

This expression can then be applied to the `operator=` of `x`:

```
template<typename TExpr>  
vector::operator=( TExpr expr ) {  
    for( size_t i = 0; i < dim; ++i )  
        this[i] = expr[i];  
}
```

Library Code:
parallelizable!

and with inlining it will be resolved to:

```
for( size_t i = 0; i < dim; ++i )  
    x[i] = a[i] * 2.0 + b[i];
```

This is done
by the
Compiler

which can then be **parallelized efficiently!**

enter for

Computing and

Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Handling of cc-NUMA architectures

- All x86-based multi-socket system will be cc-NUMA!
 - Current Operating Systems apply first-touch placement
 - If cc-NUMA is ignored, the speedup will be zero, typically
- STL provides the concept of an allocator to encapsulate memory management
 - build on the same concept to optimize for cc-NUMA
- We created two allocators:
 - Can optionally be plugged into our data types
 - `dist_allocator`: Distribute data according to OpenMP schedule type (same scheduling as in computation)
 - `chunked_allocator`: Distribute data according to explicitly precalculated scheme to improve load balancing

16

Data Type Design

- We extended the data types by adding
 - Two (optional) template parameters:
 - `Parallelization`: parallelization strategy
 - `Alloc`: STL-type allocator (cc-NUMA optimization)
 - One (optional) constructor argument:
 - `Scheduling`: Specify how work is distributed to the threads
- Abstract data type for vector + CRS-matrix implementation to
 - Evaluate Parallelization Paradigms
 - Hide Parallelization from the User
- Our approach is easy to use and delivers full performance
 - while hiding the parallelization from the user
 - and hiding cc-NUMA tuning from the user.

17

Center for

Computing and

Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

DROPS: Parallel Iteration Loop of CG-type solver

```
typedef VectorBaseCL<double, OpenMPInternalPar> VectorCL;  
  
PCG(const MatrixCL& A, VectorCL& x, const VectorCL& b,  
    const PreCon& M, int& max_iter, double& tol)  
{  
    VectorCL q(n), p(n), r(n);  
    [...]  
    for (int i=1; i<=max_iter; ++i) {  
        [...]  
        q = A * p;  
        double alpha = rho / (p*q);  
        x += alpha * p;  
        r -= alpha * q;  
        [...]  
    }  
}
```

Expression Templates allow parallelization of whole line. Here: Complete Parallel Region inside operator calls.

18

enter for

Computing and

Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Evaluation Result (1/2)

1. Library use of parallel data types (really hide parallelization):
 - OpenMP with Internal Parallelization
 - Each operator contains a distinct parallel region
 - Safe to use and completely invisible
 - TBB with Algorithms
 - Algorithmic skeletons create implicit Tasks inside operator call
 - Synchronization done by skeleton constructs
 - Safe to use and completely invisible
 - TBB with Tasks
 - Task-based Worksharing inside operator call
 - Synchronization has to be done by the library
 - Safe to use and completely invisible

19

Evaluation Result (2/2)

2. Usage to provide algorithms via library:
 - OpenMP with External Parallelization
 - Worksharing inside operator call, Parallel Region outside
 - Barriers can be eliminated by using *nowait* on Worksharing
 - Performance improvement over Internal version, but unsafe
 - OpenMP with Tasks
 - Task-based Worksharing inside operator call, Par. Reg. Outside
 - Safe, but no compiler was able to compile our code yet
- Our approach:
 - Use *safe* version during development and for algorithmic experiments
 - Carefully use possibly *unsafe* but *faster* version for production

20

Agenda

- Motivation and Computational Task
- Overview of Parallelization Paradigms
- Programmability Evaluation
- **Performance and Scalability Evaluation**
- Conclusion and Future Work

21

Center for

Computing and
Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Sequential Performance Measurements

- UMA: 2-socket quad-core Intel Xeon E5450 (3.0 GHz)
- cc-NUMA: 4-socket dual-core AMD Opteron 875 (2.2 GHz)
- Sparse Matrix-Vector-Multiplication performance [MFLOP/s]
 - Each results is measured with two threads!

Machine	Dataset	laperf (OMP) std Allocator	laperf (OMP) dist Allocator	laperf (OMP) chunked Allocator	Intel MKL
UMA	medium	406	-	-	397
UMA	large	390	-	-	383
cc-NUMA	medium	-	310	389	177
cc-NUMA	large	-	329	428	182

- Same performance as Intel MKL: Performance costs of abstractions and OpenMP parallelization are negligible
- Intel MKL has no notion / support for cc-NUMA architectures

22

Center for

Computing and
Communication

Motivation

Paradigms

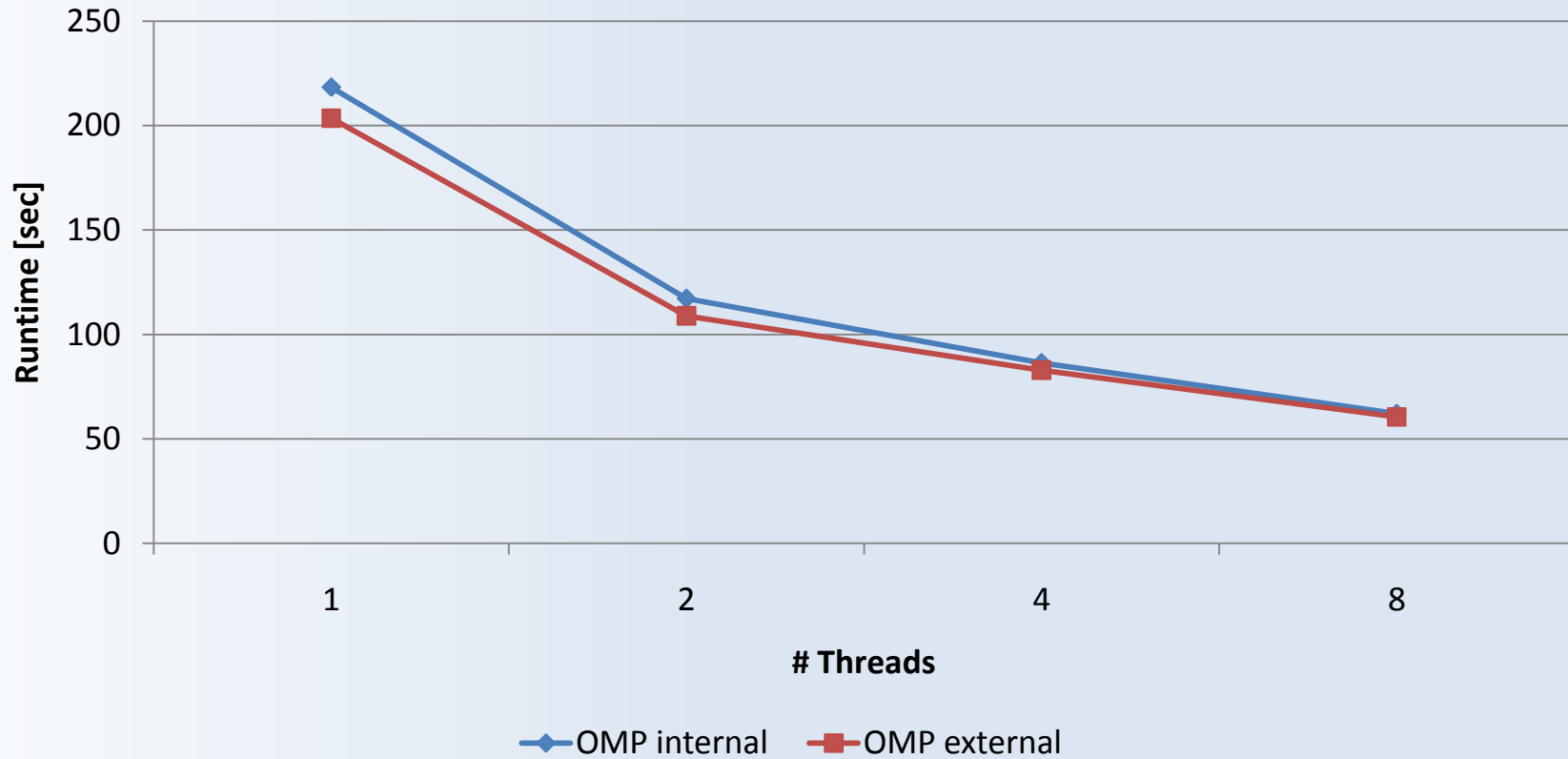
Programmability

Scalability

Conclusion

Parallel Performance Measurements (1/2)

- **Library-parallelized** GMRES solver on UMA machine:



- Scalability on UMA architecture is limited
- External Parallelization is faster (but irrelevant with 8 threads)

23

Center for

Computing and

Communication

Motivation

Paradigms

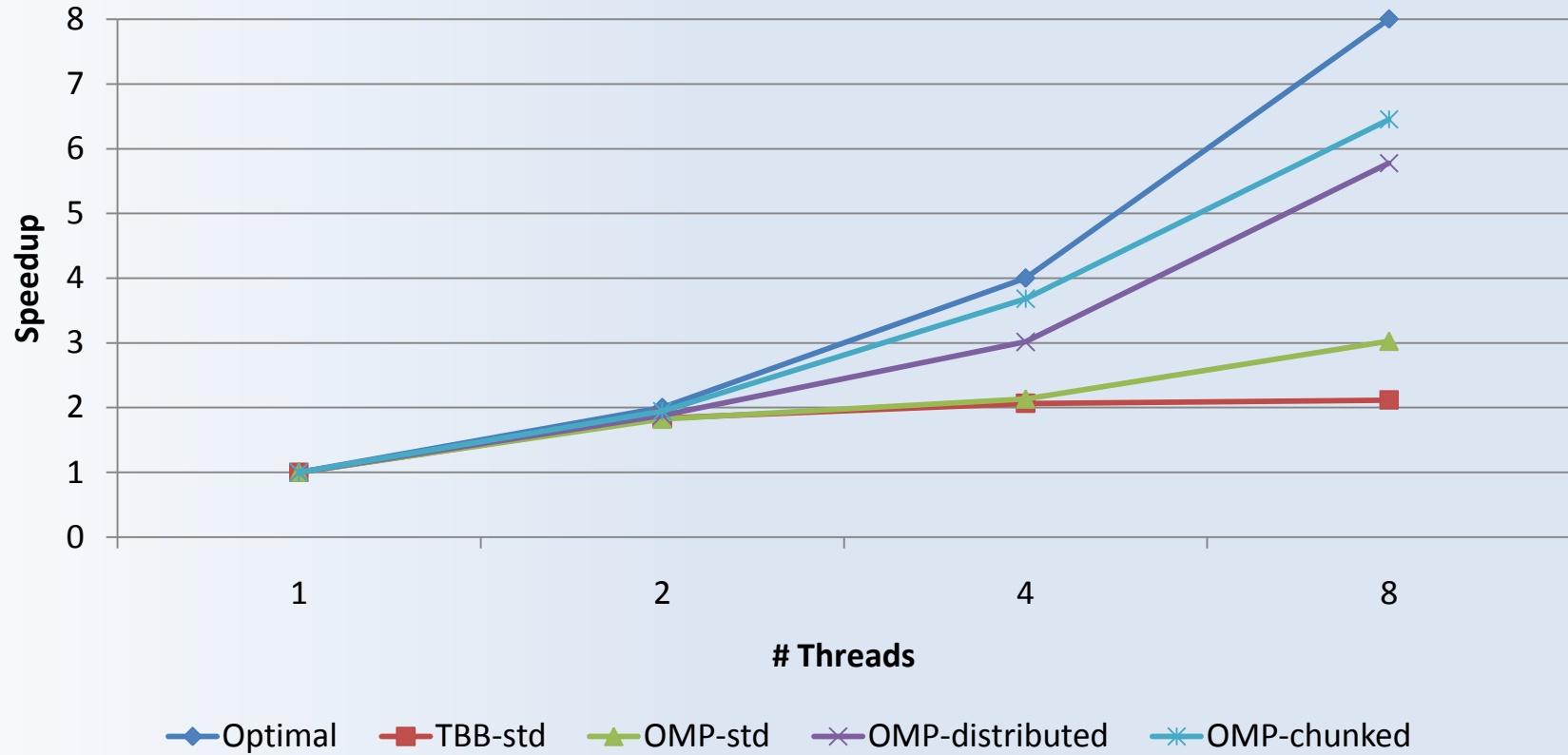
Programmability

Scalability

Conclusion

Parallel Performance Measurements (2/2)

○ **Library-parallelized** GMRES solver on cc-NUMA machine:



- cc-NUMA architecture provides good memory bandwidth
- Allocator concept successfull, TBB Tasks have no affinity

Agenda

- Motivation and Computational Task
- Overview of Parallelization Paradigms
- Programmability Evaluation
- Performance and Scalability Evaluation
- Conclusion and Future Work

25

Center for

Computing and
Communication

Motivation

Paradigms

Programmability

Scalability

Conclusion

Conclusion

- Object-oriented abstractions can be exploited to hide parallelization from the user (as much as wanted)
- Expression Templates can be used to implement parallelization efficiently
- Today's best compromise: Use OpenMP in operator functions
- Architecture abstractions proved to be easy and successful
- Future Work:
 - Apply concepts to other programming languages (FORTRAN)
 - Further exploit enhanced application knowledge
 - Provide insight for compiler / Apply optimization under the hood
 - Find a way to include aspects of Parallelization in interface descriptions of software components

The End

Thank you for
your attention!

27