Object-Oriented OpenMP Programming with C++ and FORTRAN

Christian Terboven, Dieter an Mey, Paul Kapinos,
Christopher Schleiden, Igor Merkulow
{terboven, anmey, kapinos, schleiden, merkulow}@rz.rwth-aachen.de

Center for Computing and Communication
RWTH Aachen University, Germany

June 16th, 2009
HPCS 2009, Kingston, ON, Canada

Agenda

- Motivation and Computational Task
- Some Aspects of OpenMP
- Implementation in C++ and FORTRAN
- Comparison: C++ versus FORTRAN
- Conclusion and Future Work

Object-Oriented and Parallel Programming

- Object-Oriented Programming is mainstream since the 90s, Parallel Programming is just about to enter mainstream.
  - Reasons for OO: Encapsulation and Modularity → Reusability
- Compute intense core of many PDE solvers consists of Krylov subspace methods. Variations exist among different programs and throughout the development process.
- Parallelization is often decoupled from ongoing development.
  - → Use of OO techniques to introduce and optimize parallelization
  - → Use of OO techniques to investigate parallelization approaches
  - → Use of OO techniques to hide complex architecture details from application / algorithm developer

C++: Iteration Loop of CG-style solver

```
MatrixCL A(rows, cols, nonzeros);
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;
  […]
```

Matrix data type

Vector data type

Sparse Matrix-Vector-Multiplication (SMXV)

Dot-Product

Vector Operations

- Code excerpt from C++ Navier-Stokes solver DROPS pretty much resembles notation found in math text books.
- Goals: Hide the parallelization as much as possible with as little overhead as possible to not hinder development.

FORTRAN: Iteration Loop of CG-style solver

```
integer *8 nz_num
double precision a(nz_num), irow(n+1), icol(nz_num)
double precision x(n), r(n), p(n), q(n), alpha
[…]
do iter=1, max_iter
  ...
  do i = 1, n
   q(i) = 0.0d0
   do j=irow(i),irow(i+1)-1; q(i)=q(i)+a(j)*p(icol(j)); end do
  end do

  alpha = 0.0d0
  do i = 1, n; alpha = alpha + p(i) * q(i); end do
  alpha = alpha / rho

  do i = 1, n; x(i)= x(i) + alpha * p(i); end do
  do i = 1, n; r(i) = r(i)- alpha * q(i); end do
  […]
```

Matrix

Vectors

Sparse Matrix-Vector-Multiplication (SMXV)

Dot-Product

Vector Operations

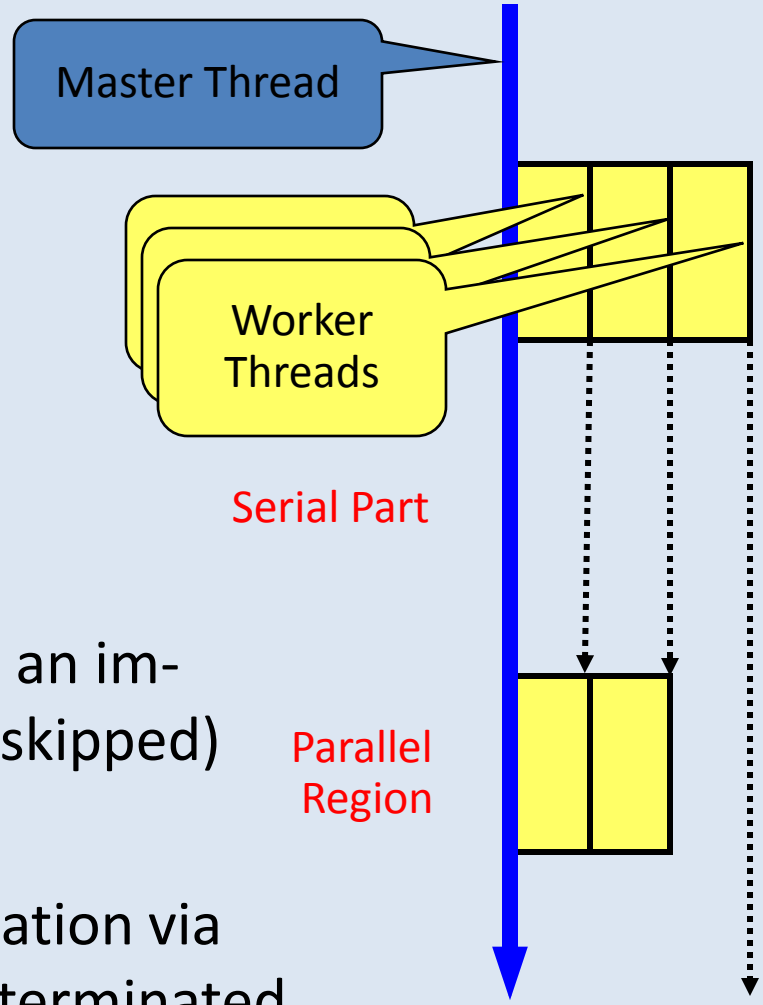
- FORTRAN: loop-oriented coding style still dominating.
- Goal: Employ object-oriented abstractions for parallelization.

Agenda

- Motivation and Computational Task
- Some Aspects of OpenMP
- Implementation in C++ and FORTRAN
- Comparison: C++ versus FORTRAN
- Conclusion and Future Work

Overview of OpenMP

- OpenMP
  - Supports Fortran, C and C++
  - Explicit Parallelization via Parallel Regions:
    - pragma + structured block
  - Worksharing
  - Task-based parallelization



- Each Worksharing construct has an implicit Barrier associated (can be skipped)
- Intel + Sun + Others: Implementation via Thread Pool → Threads are not terminated

Agenda

- Motivation and Computational Task
- Some Aspects of OpenMP
- Implementation in C++ and FORTRAN
- Comparison: C++ versus FORTRAN
- Conclusion and Future Work

C++: Naive Approach to Parallelization (Loops)

```
MatrixCL A(rows, cols, nonzeros);
VectorCL q(n), p(n), r(n);
[…]
for (int i = 1; i <= max_iter; ++i)
{
  […]
  q = A * p;
  double alpha = rho /
  x += alpha * p;
  r -= alpha * q;
  […]
```

Option 1: Replace operator calls by loops:

```
#pragma omp parallel for
for (int r = 0; r < numRows, r++)
{
   double sum = 0.0; size_t nz;
   size_t rb = Arow[r];
   size_t re = Arow[r + 1];
   for (nz = rb, nz < re, nz++)
      sum += Aval[nz] * x[Acol[nz]];
   y[r] =sum;
}
```

- Refactoring code into OpenMP loops breaks OO paradigm.
- Code changes can easily break the parallelization! (races)
- Option 2: Put parallelization inside operator calls

FORTRAN: Loop-level Parallelization

```
  do iter=1, max_iter
    […]
!$omp parallel do private(i,j)
    do i = 1, n
     q(i) = 0.0d0
     do j=irow(i),irow(i+1)-1; q(i)=q(i)+a(j)*p(icol(j)); end do
    end do
!$omp end parallel do
    alpha = 0.0d0
!$omp parallel do reduction(+:alpha)
     do i = 1, n; alpha = alpha + p(i) * q(i); end do
!$omp end parallel do
    alpha = alpha / rho
    […]
```

Sparse Matrix-Vector-Multiplication (SMXV)

Dot-Product

- Code changes can easily break the parallelization! (races)
- Possible performance improvements by merging adjacent Parallel Regions into a single (large) Parallel Region.

FORTRAN: Parallelization via WORKSHARE

```
  do iter=1, max_iter
    […]
!$omp parallel workshare
    forall (i = 1:n)
      q(i) = dot_product(a(irow(i):(irow(i+1)-1)),
                         p(icol(irow(i):(irow(i+1)-1))))
    end forall
!$omp end parallel workshare
!$omp parallel workshare
    alpha = alpha / dot_product(p,q)
!$omp end parallel workshare
!$omp parallel workshare
    x = x + alpha * p
    r = r - alpha * q
!$omp end parallel workshare
    […]
```

Sparse Matrix-Vector-Multiplication (SMXV)

Dot-Product

Vector Operations

- Array-syntax makes code much more readable, but compiler support for WORKSHARE is still weak (see performance).

C++: Parallel Matrix & Vector class

- Extend existing abstractions to introduce parallelism!
- We created our own implementation of a Matrix & Vector class and introduced them into DROPS by exchanging

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

with

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

element data type

parallelization type

- Possible problems:
  - Temporaries in complex expressions
  - Overhead introduced by the Parallelization in each operator call

C++: Matrix & Vector class with Template Expressions

- Problem: The compiler translates the user code

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

  into the following code

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

- Solution: Use Expression Templates to transform this into

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

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

- which can be efficiently parallelized with OpenMP.

FORTRAN: Parallel Matrix & Vector class (1/2)

- Matrix & Vector class in FORTRAN:

```
MODULE matvect
 TYPE :: vector ! Vector type
   DOUBLE PRECISION, ALLOCATABLE :: v(:)
 END TYPE vector
 TYPE :: matcrs ! Matrix type
   INTEGER :: n, nz_num
   INTEGER,ALLOCATABLE :: irow(:),icol(:)
   DOUBLE PRECISION, ALLOCATABLE :: a(:)
 END TYPE matcrs
 INTERFACE ASSIGNMENT (=)
   MODULE PROCEDURE assign_VV  ! vector assignment
 END INTERFACE ASSIGNMENT (=)
```

Wrapper around native vector

Implemented via allocatables

- Using Pointers *would* be more flexible, but slower. Fortran 2003 *would* bring more comfort, but is not yet supported by OpenMP. Compiler *would* provide slow, serial assignment.

FORTRAN: Parallel Matrix & Vector class (2/2)

- Similar approach as used in C++ library implementation: Complete Parallel Region inside of the operators

```
SUBROUTINE assign_VV (left, right)
  TYPE(vector),INTENT(OUT) :: left
  TYPE(vector),INTENT(IN)  :: right
  INTEGER :: i
  IF(.NOT.ALLOCATED(left%v))
     ALLOCATE(left%v(SIZE(right%v)))
!$omp parallel do schedule(runtime)
  do i = 1, SIZE(right%v); left%v(i)=right%v(i); end do
!$omp end parallel do
END SUBROUTINE assign_VV
```

- No problem with temporaries in FORTRAN as everything is allocated explicitly – multiple Parallel Regions in compound operations, though.

Handling of cc-NUMA architectures

- All x86-based multi-socket systems will be cc-NUMA!
  - Current Operating Systems apply first-touch placement
- C++: STL provides the concept of an allocator to encapsulate memory management → extend that concept to optimize for cc-NUMA
  - `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 balacing
- FORTRAN: No support for class constructors yet, but an explicit *initialization* function could fulfill the purpose.
- C++ and FORTRAN: Thread binding is a necessity on cc-NUMA

C++: Parallel Iteration Loop of CG-style solver

```
MatrixCL<OpenMPParallelization> A(rows, cols, nonzeros);
VectorCL<OpenMPParallelization> 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;
  […]
```

Enable Parallelization

Complete Parallel Region

- Expression Templates allow the parallelization of whole lines.
- Parallelization is completely invisible, algorithmic modifications do not break the parallelization!

FORTRAN: Parallel Iteration Loop of CG-style solver

```
USE matvect
  TYPE(matcrs) :: a
  TYPE(vector) :: x, r, p, q
  DOUBLE PRECISION :: alpha
  […]
  do iter=1, max_iter
    […]
    q = a * p

    alpha = norm_2(p,q)

    x = x + alpha * p
    r = r - alpha * q
    […]
```

Use parallel data types

3 separate Parallel Regions

- Parallelization is completely invisible, algorithmic modifications do not break the parallelization!

Agenda

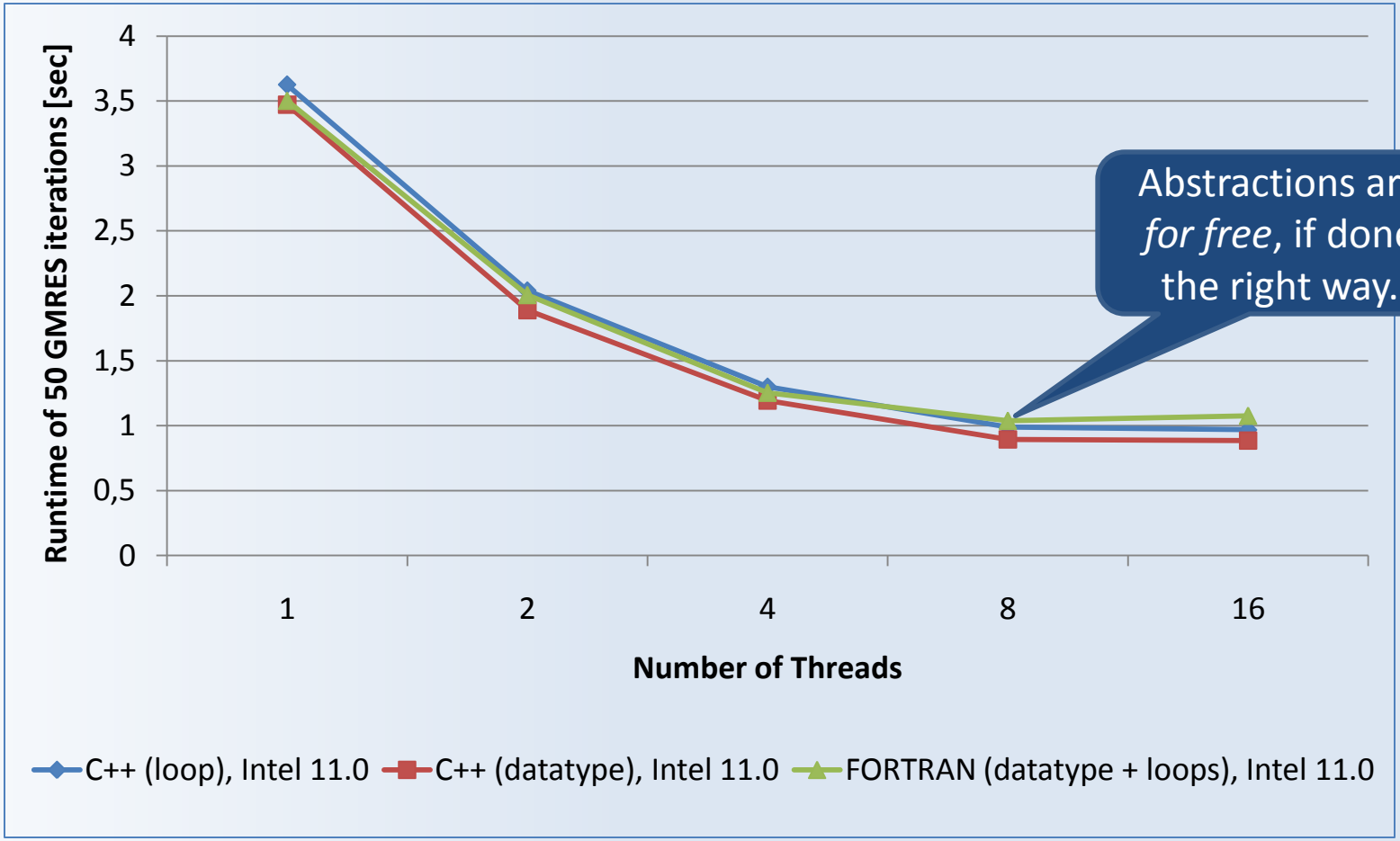
- Motivation and Computational Task
- Some Aspects of OpenMP
- Implementation in C++ and FORTRAN
- Comparison: C++ versus FORTRAN
- Conclusion and Future Work

Comparison: Programmability

- OpenMP with Internal Parallelization (self-contained Parallel Region per operator): Completely invisible and safe to use.
  - Faster External Parallelization - employ orphaned Work-sharing constructs in operator, Parallel Region is outside – can easily lead to data races.
- C++: Object-oriented design allows to write parallel code that still resembles math text book notation → existing data type design does not have to be broken.
- FORTRAN: Array-syntax clearly improves readability, but full object-oriented design is achievable as well.

Comparison: Performance (Big Picture)

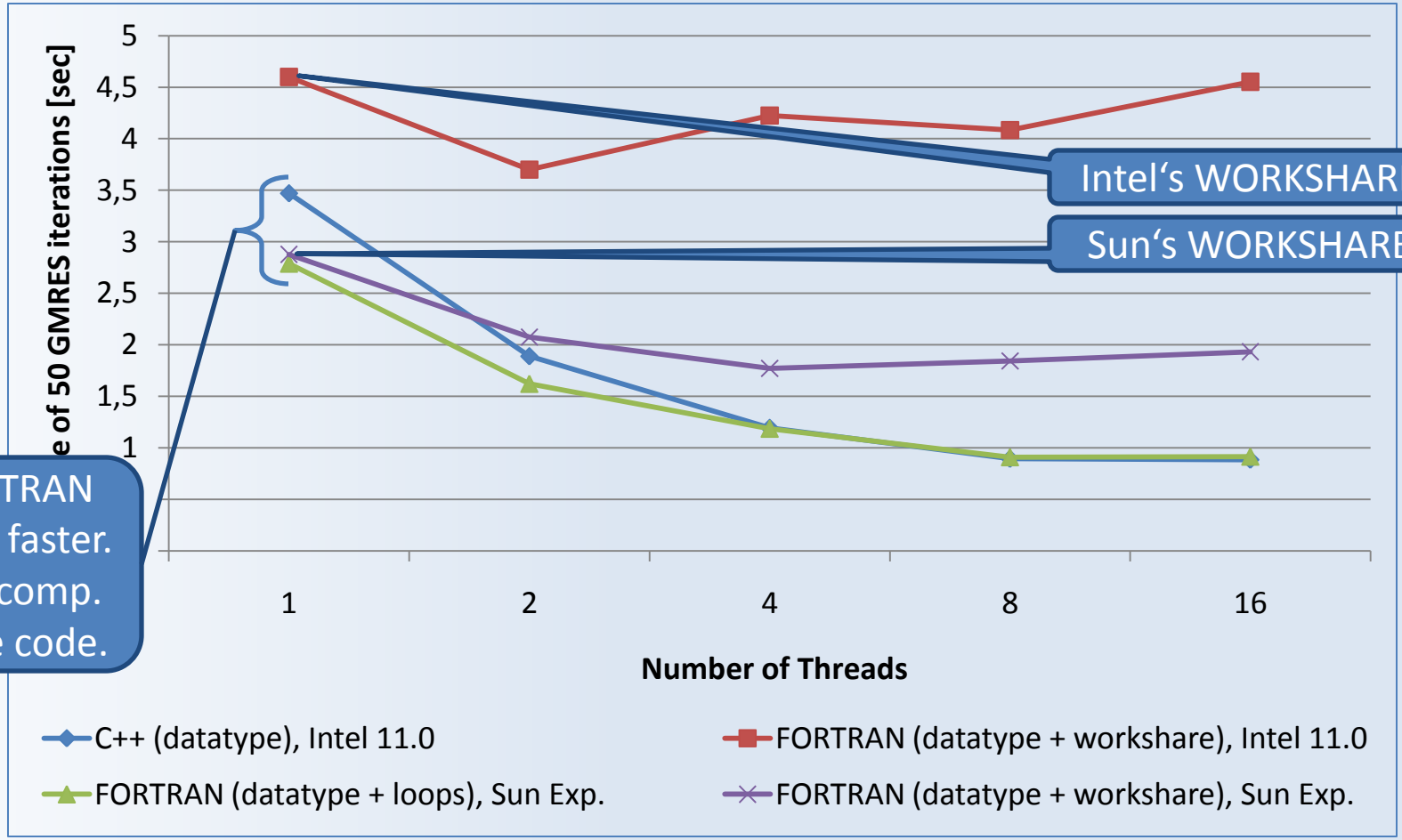
- Intel Compiler: hardly any difference in the programming language.



Two-socket Intel Nehalem (X5570) @ 2.93 GHz, Intel 11.0 compiler.

Comparison: Performance (Details)

- Quality of FORTRAN implementations varies.



Sun‘s FORTRAN compiler is faster. Sun‘s C++ comp. fails on the code.

Two-socket Intel Nehalem (X5570) @ 2.93 GHz

Agenda

- Motivation and Computational Task
- Some Aspects of OpenMP
- Implementation in C++ and FORTRAN
- Comparison: C++ versus FORTRAN
- Conclusion and Future Work

Conclusion

- Object-oriented abstractions can be exploited to hide parallelization from the user (as much as wanted).
- C++: Expression Templates can be used to implement parallelization very efficiently, but with lot of coding work.
- Todays best compromise: Use OpenMP in operator functions.

- Future Work:
  - Eliminate unneccessary barriers
  - Further leverage enhanced application knowlegde
    - Provide insight for compiler / Apply optimization under the hood
  - Identify FORTRAN 2003 features which need to be supported by OpenMP
  - Find a way to include aspects of parallelization in interface descriptions of software components

The End

Thank you for

your attention!