Exploiting Object-Oriented Abstractions to parallelize Sparse Linear Algebra Codes

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

September 3rd, 2009
ParCo 2009, Lyon, France
Exploiting OO-Abstractions to parallelize Sparse LA Codes

03.09.2009 – C. Terboven

Agenda

- Motivation and Computational Task
  - Implementation in C++
  - Implementation in FORTRAN
  - Comparison: C++ versus FORTRAN
  - Conclusion and Future Work
Object-Oriented and Parallel Programming

- Compute intense core of many PDE solvers consists of Krylov subspace methods. Variations exist among different programs and throughout the development process.

- Object-Oriented Programming is mainstream since the 90s, Parallel Programming is just about to enter mainstream.
  - Reasons for OO: Encapsulation and Modularity → Reusability

- 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

Motivation
Motivation

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;
    […]

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.
Aspects 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
- Implementation in C++
- Implementation in 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 / (p * q);
    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)
C++: Parallel Matrix & Vector class

- Extend existing abstractions to introduce parallelism!
- DROPS: Numerical parts are implemented via Matrix (CRS) and Vector class, or descendents of those.
  - Use Adapter design pattern to replace those with parallel ones

Scheme showing integration of parallel classes into DROPS.

Problems:
- Temporaries in complex expressions.
- Overhead introduced by the Parallel Region in every operator call.
Exploiting OO-Abstractions to parallelize Sparse LA Codes

C++: Matrix & Vector class with Template Expressions

- Problem: The compiler translates the user code
  \[ x = (a \times 2.0) + b; \]
  into the following code
  \[
  \text{laperf::vector<double> } _t1 = \text{operator\ast}(a, 2.0);
  \text{laperf::vector<double> } _t2 = \text{operator\ast}(\_t1, b);
  x.\text{operator\ast}(\_t2);
  \]

- Solution: Use Expression Templates to transform this into
  \[
  \text{LB<OpAdd, LB<OpMul, vector, double>, vector>}
  \text{expr< LB<OpAdd, LB<OpMul, vector, double>, vector>(}
  \text{LB<OpMul, vector, double>(a,2.0), b}
  \text{)}>;
  \]
  \[
  \text{template<typename TExpr>}
  \text{vector::operator\ast( TExpr expr ) } \{
  \text{#pragma omp parallel for}
  \text{for( size_t } i = 0; i < \text{dim}; ++i )}
  \text{this[i] = expr[i];}
  \}
  \]
  which can be efficiently parallelized with OpenMP.

... OpenMP ...

Motivation  C++  FORTRAN  Comparison  Conclusion
Handling of cc-NUMA architectures

- All x86-based multi-socket systems will be cc-NUMA!
  - Current Operating Systems apply first-touch placement

- Thread binding is a necessity on cc-NUMA system: Hardware is examined automatically at application startup (*Singleton* design pattern).

- Policy-based approach: Apply the *Strategy* design pattern to influence the internal workings of a target class.

- Provide simple-to-choose-from options for the user:
  - *DistributedPolicy*: Distribute data according to OpenMP schedule type (same scheduling as in computation, default)
  - *ChunkedPolicy*: Distribute data according to explicitly precalculated scheme to improve load balancing (special cases)
C++: Parallel Iteration Loop of CG-style solver

```c++
MatrixCL 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;
    […]
```

- Expression Templates allow the parallelization of whole lines.
- Parallelization is completely invisible, algorithmic modifications do not break the parallelization!
- Incremental approach: (i) Go Parallel, (ii) cc-NUMA, (iii) ...

Enable Parallelization
Only one Parallel Region
Agenda

- Motivation and Computational Task
- Implementation in C++
- Implementation in FORTRAN
- Comparison: C++ versus FORTRAN
- Conclusion and Future Work
FORTRAN: Loop-level and WORKSHARE Parallelization

- Sparse Matrix-Vector-Multiplication (SMXV) with loops:
  ```fortran
  !$omp parallel do private(i,j)
  do i = 1, n
    q(i) = 0.0d0
    do j=irow(i),irow(i+1)-1; q(i)=q(i)+a(j)*p(icol(j)); end do
  end do
  !$omp end parallel do
  ```

- Code changes can easily break the parallelization! (races)

- Dot-Product with WORKSHARE:
  ```fortran
  !$omp parallel workshare
  alpha = alpha / dot_product(p,q)
  !$omp end parallel workshare
  ```

- Array-syntax makes code much more readable, but compiler support for WORKSHARE is still weak (see performance).
Matrix & Vector class in FORTRAN:

```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 (=)
```

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 the operators:

  ```fortran
  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
  ```

- Problem of temporaries in complex expressions cannot be solved in FORTRAN. We did not see any compiler aggressive enough to optimize these temporaries away.
Exploiting OO-Abstractions to parallelize Sparse LA Codes

FORTRAN: **Parallel** Iteration Loop of CG-style solver

```fortran
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
[...]
```

- Parallelization is completely invisible, algorithmic modifications do not break the parallelization!
- More Overhead than in C++ implementation.
Exploiting OO-Abstractions to parallelize Sparse LA Codes

03.09.2009 – C. Terboven

Agenda

- Motivation and Computational Task
- Implementation in C++
- Implementation in 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.
  - External Parallelization - employ orphaned Worksharing constructs in operator, Parallel Region is outside – could be faster but can easily lead to data races.

- C++: Object-oriented design allows to write parallel code that still resembles math text book notation, parallel data types can easily be integrated into existing application → minimal changes to user application.

- 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.

Abstractions are *for free*, if done the right way.

Two-socket Intel Nehalem (X5570) @ 2.93 GHz, Intel 11.0 compiler.
Quality of FORTRAN implementations varies.

Motivation

C++

FORTRAN

Comparison

Conclusion

Two-socket Intel Nehalem (X5570) @ 2.93 GHz

Sun’s FORTRAN compiler is faster. Sun’s C++ comp. fails on the code.
cc-NUMA optimization became simple, but is very important!

Our library’s default binding ensures steady increase in performance.

Difference of ignoring or taking cc-NUMA into account!

Quad-socket AMD Barcelona (8356) @ 2.30 GHz
Exploiting OO-Abstractions to parallelize Sparse LA Codes

03.09.2009 – C. Terboven

Agenda

- Motivation and Computational Task
- Implementation in C++
- Implementation in 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), lowering the burden of parallelization and optimization.

- C++: Expression Templates can be used to implement parallelization very efficiently, but with some coding work. FORTRAN works as well.

- Best compromise: Use OpenMP inside operators.

- Future Work:
  - Further leverage abstractions in other domains (i.e. Mesh)
  - Identify FORTRAN 2003 features which need to be (better) supported by OpenMP
  - Composability: Include details of the parallelization in interface descriptions of software components
The End

Thank you for your attention!