Experiences with the OpenMP Parallelization of DROPS, a Navier-Stokes-Solver written in C++

Christian Terboven, Alexander Spiegel, Dieter an Mey
Center for Computing and Communication, RWTH Aachen University, Germany
{Terboven|Spiegel|anMey}@rz.rwth-aachen.de

Sven Groß, Volker Reichelt
Institut für Geometrie und Praktische Mathematik, RWTH Aachen University, Germany
{Gross|Reichelt}@igpm.rwth-aachen.de

Outline

- The DROPS multi-phase Navier-Stokes solver
- Portability and Performance of the Serial Program Version
- The OpenMP Approach
- Performance of the OpenMP Version
- Summary

1. **The DROPS multi-phase Navier-Stokes solver**
2. Portability and Performance of the Serial Program Version
3. The OpenMP Approach
4. Performance of the OpenMP Version
5. Summary

Introduction

- DROPS is developed in the context of SFB 540 TP B4
  Head: Prof. Dr. Arnold Reusken
  (SFB=long-term multi-disciplinary collaborative research center for universities,
  TP = project within an SFB)

- SFB 540: Model-based Experimental Analysis of Kinetic Phenomena in Fluid Multi-phase Reactive Systems
  http://www.sfb540.rwth-aachen.de/

- TP B4: Multigrid Methods for the Numerical Simulation of Reactive Multiphase Fluid Flow Models
  http://www.sfb540.rwth-aachen.de/Projects/tpb4.php

- DROPS is a solver for the incompressible Navier-Stokes equations for the numerical simulation of two-phase fluid flow models
  (in development)

- Target of this work: serial tuning and OpenMP parallelization

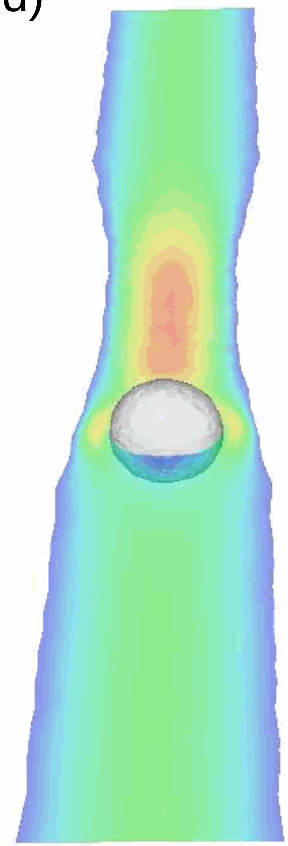
RWTH Center for Computing and Communication

DROPS

- Numerical simulation of two-phase flow
- The two-phase flow is modeled by the instationary and non-linear Navier-Stokes equation
- So-called level set function is used to describe the interface between the two phases
- DROPS is written in C++
  (object-oriented, templates, STL, compile-time polymorphism)

- Adaptive Tetrahedral Grid Hierarchy
- Finite Element Method (FEM)

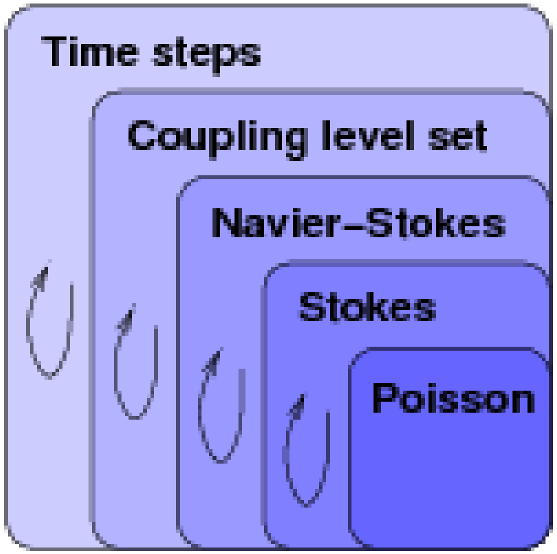
Example:
Silicon oil drop in D_2O
(fluid/fluid)



RWTH Center for Computing and Communication

Nested Solvers

- Time integration by fractional step method
- Fixed point iteration for the decoupled Navier-Stokes and the advection equations for the level set function
- Fixed point iteration for non-linear convection term in the Navier-Stokes equations
- Stokes solvers: Uzawa, Schur, MinRes, **GMRES**
- Inner solvers for Poisson-type problems.
  - preconditioned conjugate gradient (**PCG**)
  - multi-grid (MG)
- Preconditioners / smoothers: **Jacobi** or **SSOR**



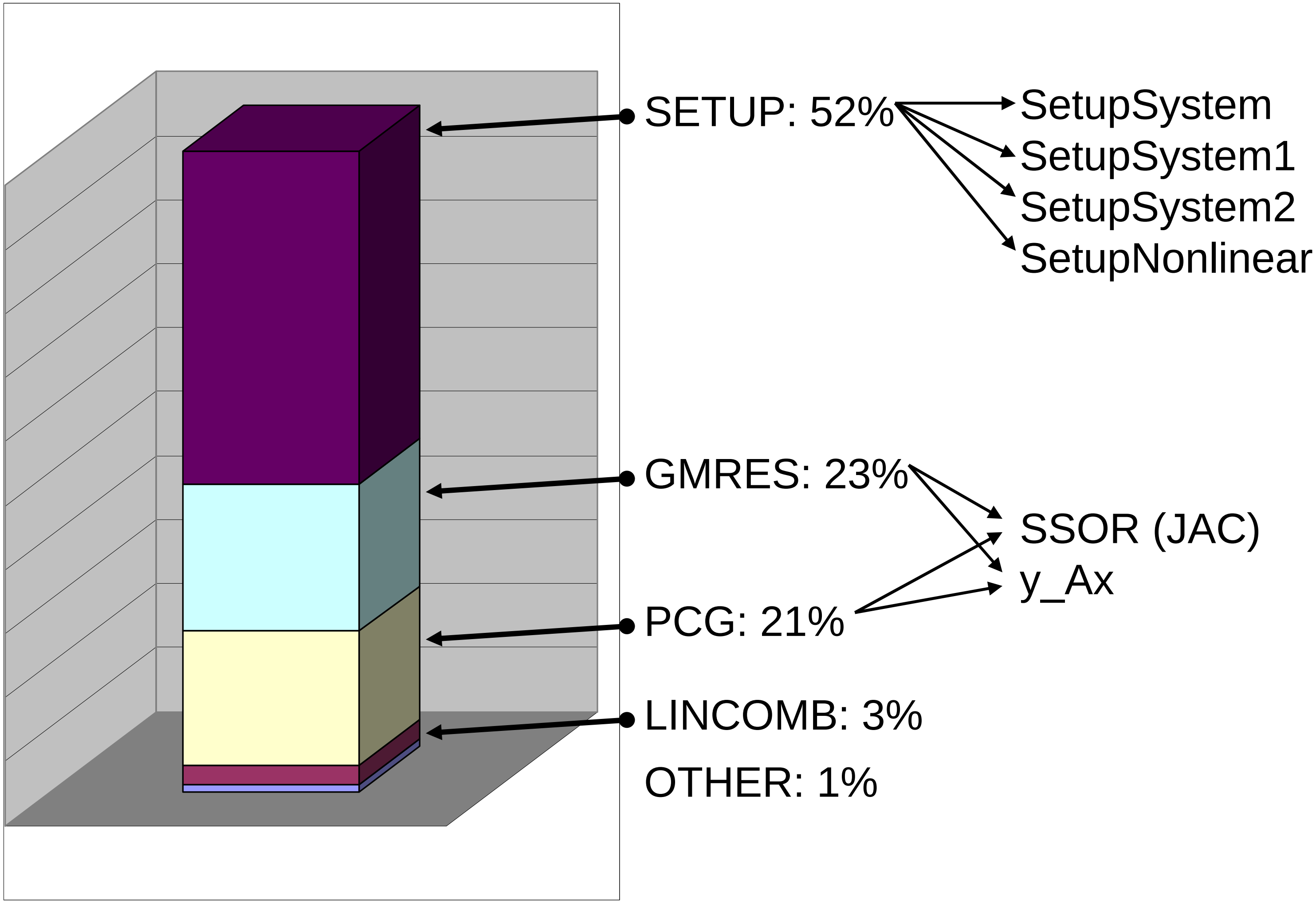
The GMRES and PCG solvers were employed and parallelized in this work

1. The DROPS multi-phase Navier-Stokes solver
2. **Portability and Performance of the Serial Program Version**
3. The OpenMP Approach
4. Performance of the OpenMP Version
5. Summary

Portability and Performance of the Serial Version Platforms

code	machine	processor	operating system	compiler
XEON+gcc333	standard PC	2x Intel Xeon, 2.66 GHz	Fedora-Linux	GNU C++ V3.3.3
XEON+gcc343	standard PC	2x Intel Xeon, 2.66 GHz	Fedora-Linux	GNU C++ V3.4.3
XEON+icc81	standard PC	2x Intel Xeon, 2.66 GHz	Fedora-Linux	Intel C++ V8.1
XEON+pgi60	standard PC	2x Intel Xeon, 2.66 GHz	Fedora-Linux	PGI C++ V6.0-1
XEON+vs2005	standard PC	2x Intel Xeon, 2.66 GHz	Windows 2003	MS Visual Studio 2005, beta 2
OPT+gcc333	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	GNU C++ V3.3.3
OPT+gcc333X	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	GNU C++ V3.3.3, 64bit
OPT+icc81	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	Intel C++ V8.1
OPT+icc81X	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	Intel C++ V8.1, 64bit
OPT+pgi60	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	PGI C++ V6.0-1
OPT+pgi60X	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	PGI C++ V6.0-1, 64bit
OPT+path20	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	PathScale EKOpath 2.0
OPT+path20X	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Fedora-Linux	PathScale EKOpath 64bit
OPT+ss10	Sun Fire V40z	4x AMD Opteron, 2.2 GHz	Solaris 10	SunStudio C++ V10
USIV+gcc331	Sun Fire E2900	12x UltraSPARC IV, 1.2 GHz, dual core	Solaris 9	GNU C++ V3.3.1
USIV+ss10	Sun Fire E2900	12x UltraSPARC IV, 1.2 GHz, dual core	Solaris 9	Sun Studio C++ V10
USIV+guide	Sun Fire E2900	12x UltraSPARC IV, 1.2 GHz, dual core	Solaris 9	Intel-KSL Guidec++ V4.0 + Sun Studio 9
POW4+guide	IBM p690	16x Power4, 1.7 GHz, dual core	AIX 5L V5.2	Intel-KSL Guidec++ V4.0
POW4+xlC60	IBM p690	16x Power4, 1.7 GHz, dual core	AIX 5L V5.2	IBM Visual Age C++ V6.0
POW4+gcc343	IBM p690	16x Power4, 1.7 GHz, dual core	AIX 5L V5.2	GNU C++ V3.3.3
IT2+icc81	SGI Altix 3700	128x Itanium 2, 1.3 GHz	SGI ProPack Linux	Intel C++ V8.1

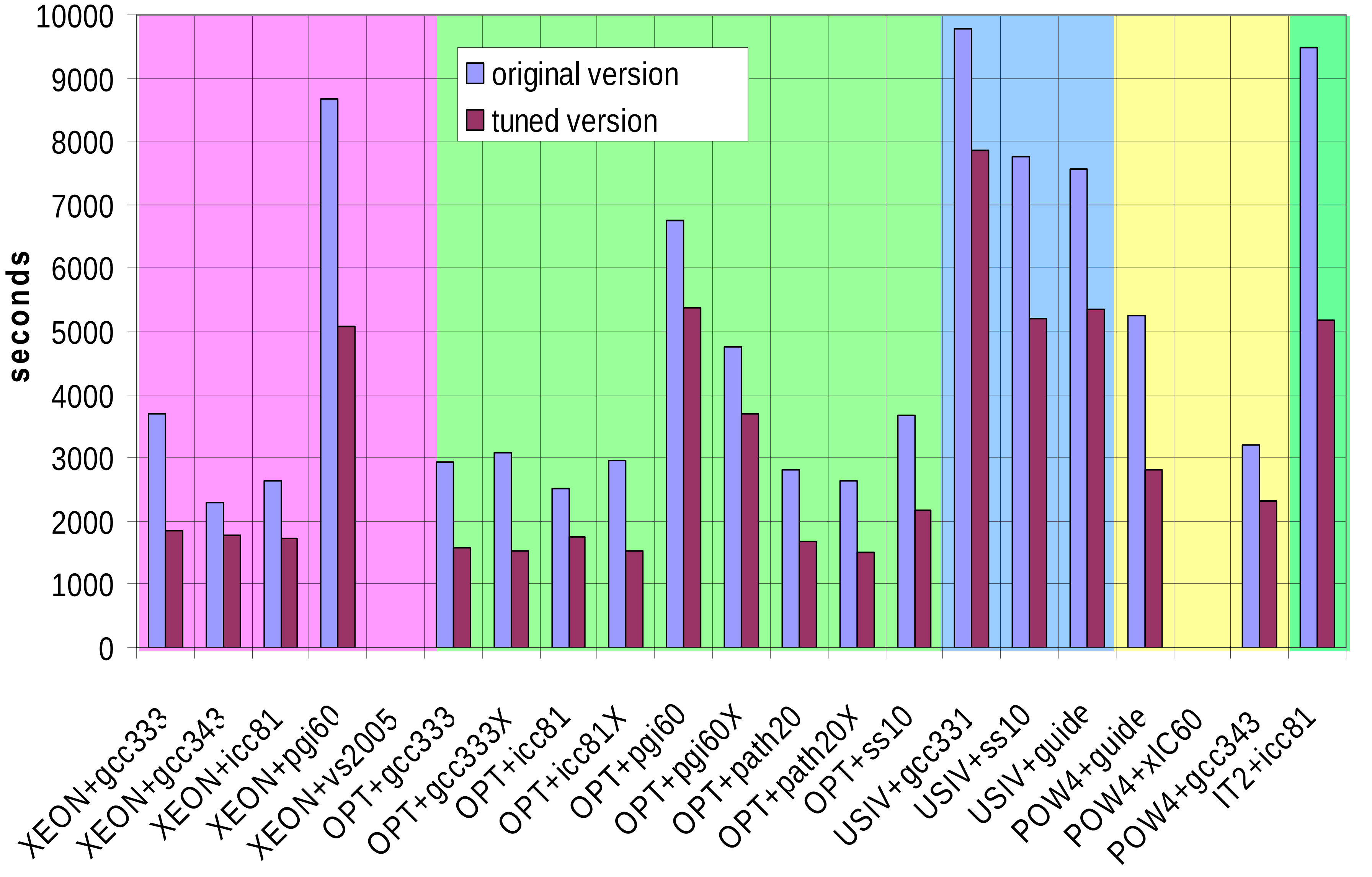
Portability and Performance of the Serial Version
Runtime Profile



Portability and Performance of the Serial Version
Serial Tuning

- **Manual and automatic prefetching**
- **Reduce usage of the maps STL containers, if the shape of the stiffness matrix does not change (leads to increased overhead of the parallel version)**
- **64 bit address mode advantageous on Opteron**
- **Note, that we didn't have exclusive access to the Power4 and Itanium2 based systems for our timing measurements**
- Most **experience** and **ambitions** on UltraSPARC- and Opteron-based systems, less experience and ambitions on Power4- and Itanium-based systems.

Portability and Performance of the Serial Version
Serial Runtime



Portability and Performance of the Serial Version
Serial Runtime

code	compiler options	runtime [s] orig. version	runtime [s] tuned version
XEON+gcc333	-O2 -march=pentium4	3694.9	1844.3
XEON+gcc343	-O2 -march=pentium4	2283.3	1780.7
XEON+icc81	-O3 -tpp7 -xN -ip	2643.3	1722.9
XEON+pgi60	-fast -tp piv	8680.1	5080.2
XEON+vs2005	compilation fails	n.a.	n.a.
OPT+gcc333	-O2 -march=opteron -m32	2923.3	1580.3
OPT+gcc333X	-O2 -march=opteron -m64	3090.9	1519.5
OPT+icc81	-O3 -ip -g	2516.9	1760.7
OPT+icc81X	-O3 -ip -g	2951.3	1521.2
OPT+pgi60	-fast -tp k8-32 -fastsse	6741.7	5372.9
OPT+pgi60X	-fast -tp k8-64 -fastsse	4755.1	3688.4
OPT+path20	-O3 -march=opteron -m32	2819.3	1673.1
OPT+path20X	-O3 -march=opteron -m64	2634.5	1512.3
OPT+ss10	-fast -features=no%except -xtarget=opteron	3657.8	2158.9
USIV+gcc331	-O2	9782.4	7845.4
USIV+ss10	-fast -xtarget=ultra4	7749.9	5198
USIV+guide	-fast +K3 -xipo=2 -xtarget=ultra4 -lmtmalloc	7551	5335
POW4+guide	+K3 -backend -qhot -backend -O3 -backend -g -bmaxdata:0x80000000]	5251.9	2819.4
POW4+xlC60	compilation fails	n.a.	n.a.
POW4+gcc343	-O2 -maix64 -mpowerpc64	3193.7	2326
IT2+icc81	-O3 -ip -g	9479	5182.8

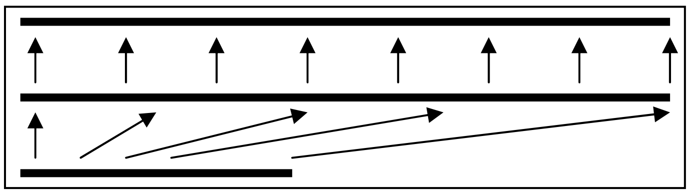
Portability and Performance of the Serial Version
Serial MFlop/s

code	compiler options	Mflop/s orig. version	Mflop/s tuned version
XEON+gcc333	-O2 -march=pentium4	63.61	125.37
XEON+gcc343	-O2 -march=pentium4	102.94	129.85
XEON+icc81	-O3 -tpp7 -xN -ip	88.92	134.20
XEON+pgi60	-fast -tp piv	27.08	45.51
XEON+vs2005	compilation fails	n.a.	n.a.
OPT+gcc333	-O2 -march=opteron -m32	80.40	146.31
OPT+gcc333X	-O2 -march=opteron -m64	76.04	152.17
OPT+icc81	-O3 -ip -g	93.38	131.32
OPT+icc81X	-O3 -ip -g	79.64	152.00
OPT+pgi60	-fast -tp k8-32 -fastsse	34.86	43.03
OPT+pgi60X	-fast -tp k8-64 -fastsse	49.43	62.69
OPT+path20	-O3 -march=opteron -m32	83.37	138.20
OPT+path20X	-O3 -march=opteron -m64	89.21	152.89
OPT+ss10	-fast -features=no%except -xtarget=opteron	64.26	107.10
USIV+gcc331	-O2	24.03	29.47
USIV+ss10	-fast -xtarget=ultra4	30.33	44.48
USIV+guide	-fast +K3 -xipo=2 -xtarget=ultra4 -lmtmalloc	31.13	43.34
POW4+guide	+K3 -backend -qhot -backend -O3 -backend -g -bmaxdata:0x80000000]	44.75	82.01
POW4+xlC60	compilation fails	n.a.	n.a.
POW4+gcc343	-O2 -maix64 -mpowerpc64	73.59	99.40
IT2+icc81	-O3 -ip -g	24.80	44.61

1. The DROPS multi-phase Navier-Stokes solver
2. Portability and Performance of the Serial Program Version
3. **The OpenMP Approach**
4. Performance of the OpenMP Version
5. Summary

The OpenMP Approach
Assembly of the Stiffness Matrices

- Since the **matrices** arising from the discretization step are **sparse**, an appropriate matrix storage format (**CRS** = compressed row storage) is used

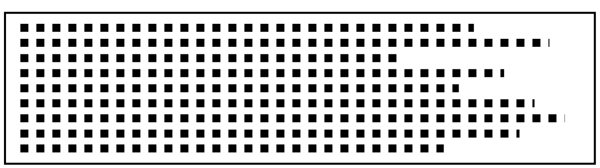


val – valarray<double> of dimension #nz

col_ind – valarray<size_t**> of dimension #nz**

row_ptr – valarray<size_t**> of dimension #rows**

- **Insertion** of elements is rather **expensive**, therefore during the **discretization** step the values are stored in an intermediate format based on map<size_t, double>.



one map<size_t, double> per row

map<key_type, data_type>: store a set of elements of type data_type, **access using a key** element of type index_type.

The OpenMP Approach
Assembly of the Stiffness Matrices

- Parallelization: the routines assembling the stiffness matrices typically use STL iterators loops like:

```
for (MultiGridCL::const_TriangTetraIteratorCL
      sit  = _MG.GetTriangTetraBegin(lvl),
      send = _MG.GetTriangTetraEnd(lvl);
      sit != send; sit++)
```

- **Such a loop construct cannot be parallelized in OpenMP, because the loop iteration variable is not of type integer**
  → store the pointers in an additional array

```
std::vector<const TetraCL*> myTetras(lSize); lPos = 0;
for (MultiGridCL::const_TriangTetraIteratorCL
      sit  = _MG.GetTriangTetraBegin(lvl),
      send = _MG.GetTriangTetraEnd(lvl);
      sit != send; sit++)
     {
          myTetras[lPos] = &*sit;   lPos++;
     }
```

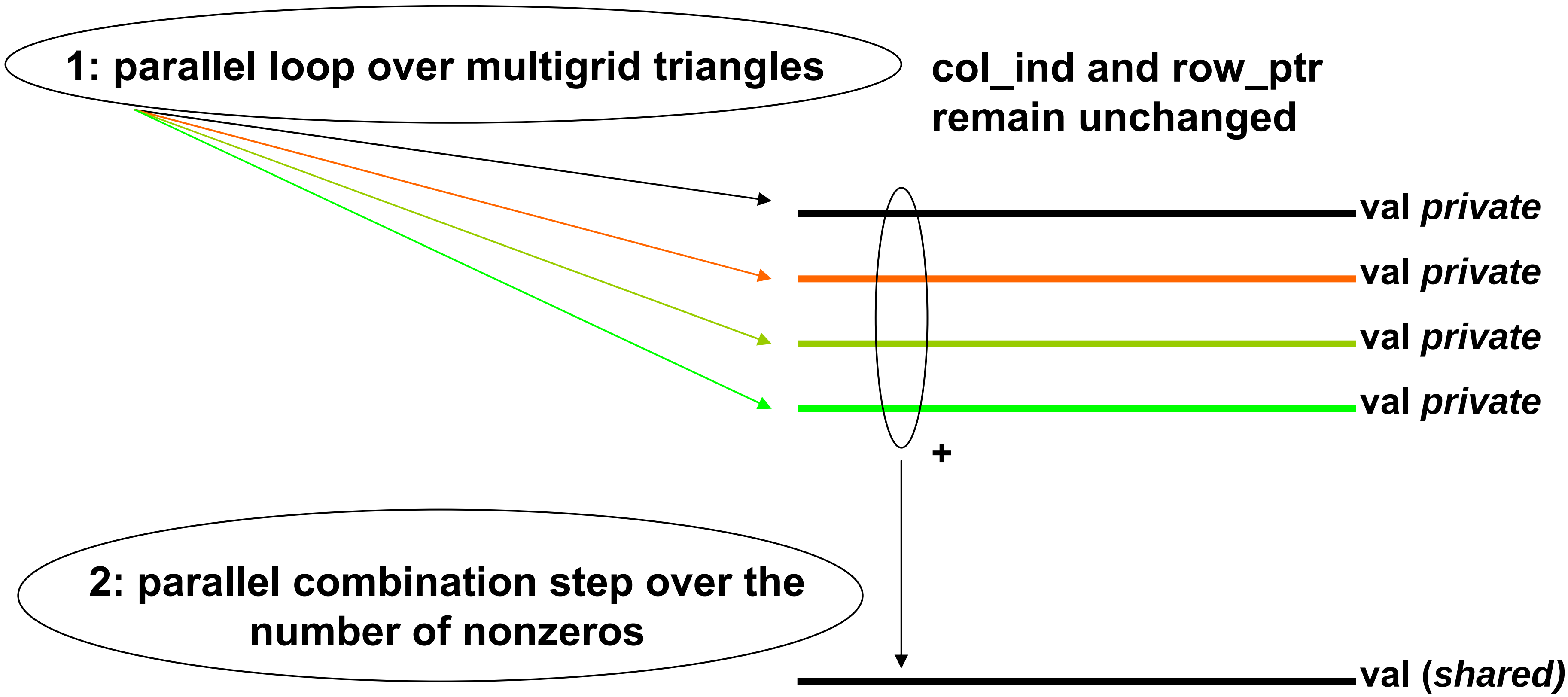
The OpenMP Approach
Assembly of the Stiffness Matrices

- Serial tuning:
  as long as the structure of the matrix does not change,
  reuse the index vectors and only fill the matrix with new data values
  -> **REUSE**

- Two versions have to be considered for parallelization:
  - **REUSE**: called many times during a time step
  - **NO-REUSE**: called only if the shape of the matrices change DROPS

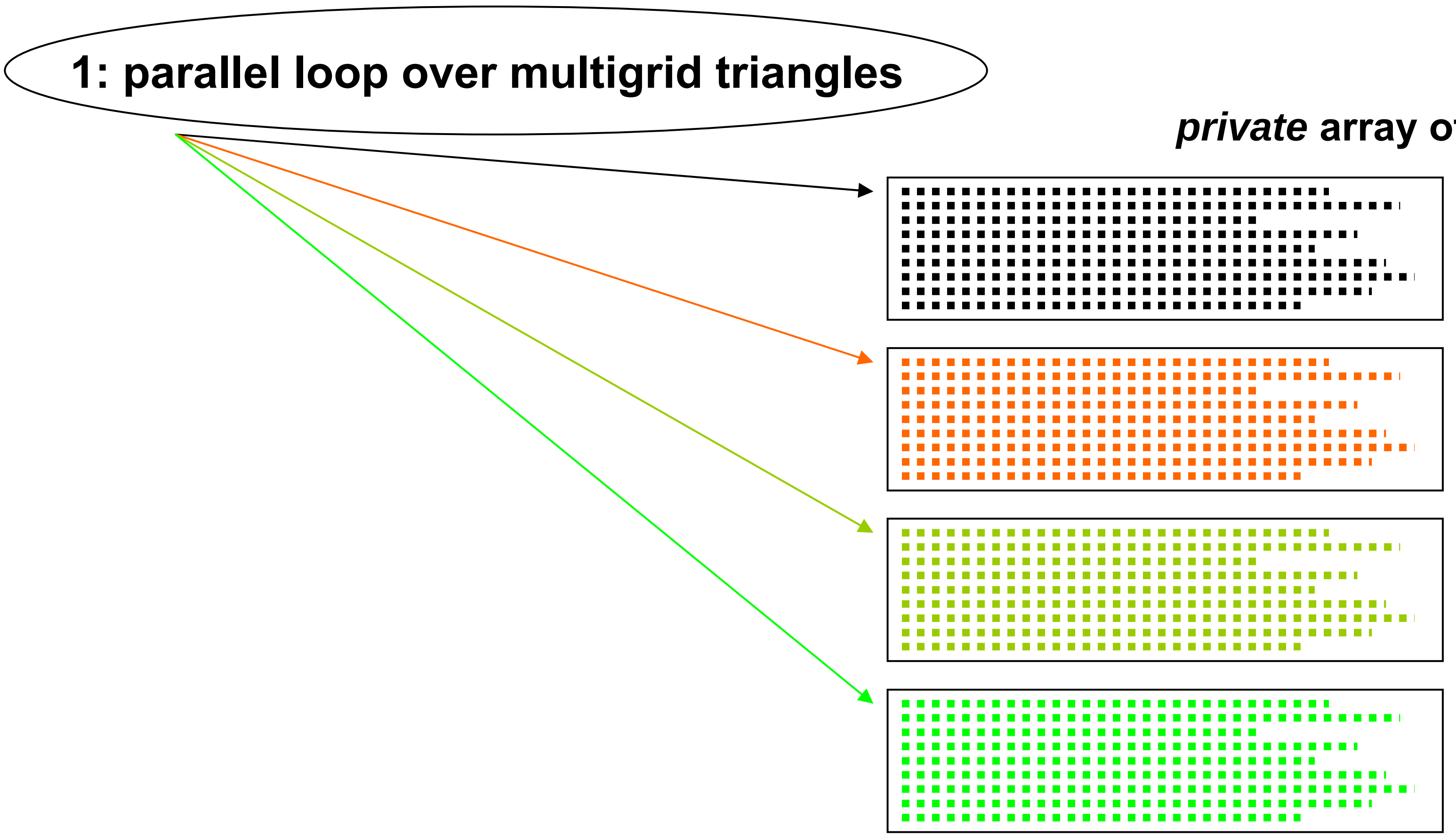
The OpenMP Approach
Assembly of the Stiffness Matrices

- Parallelization in **REUSE-mode** (example with four threads):

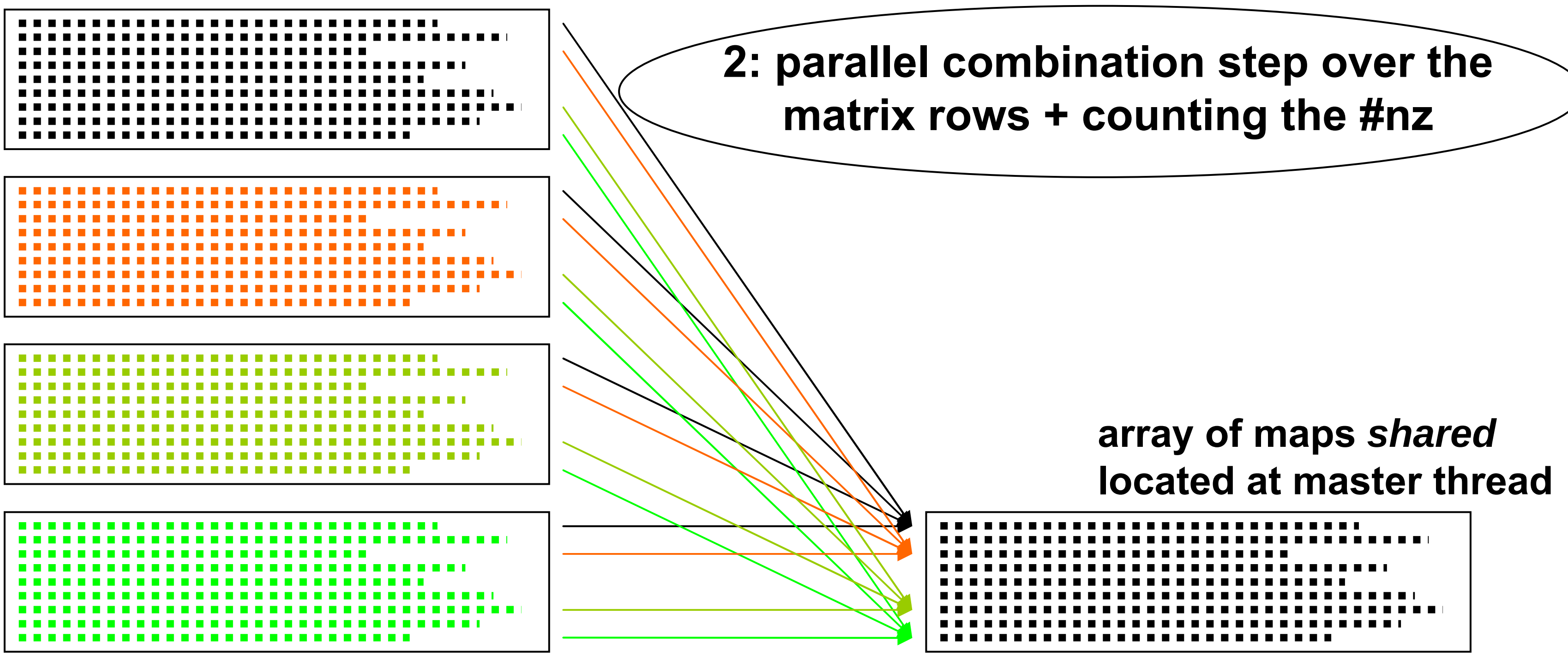


The OpenMP Approach
Assembly of the Stiffness Matrices

- Parallelization in **NO-REUSE-mode** (example with four threads):



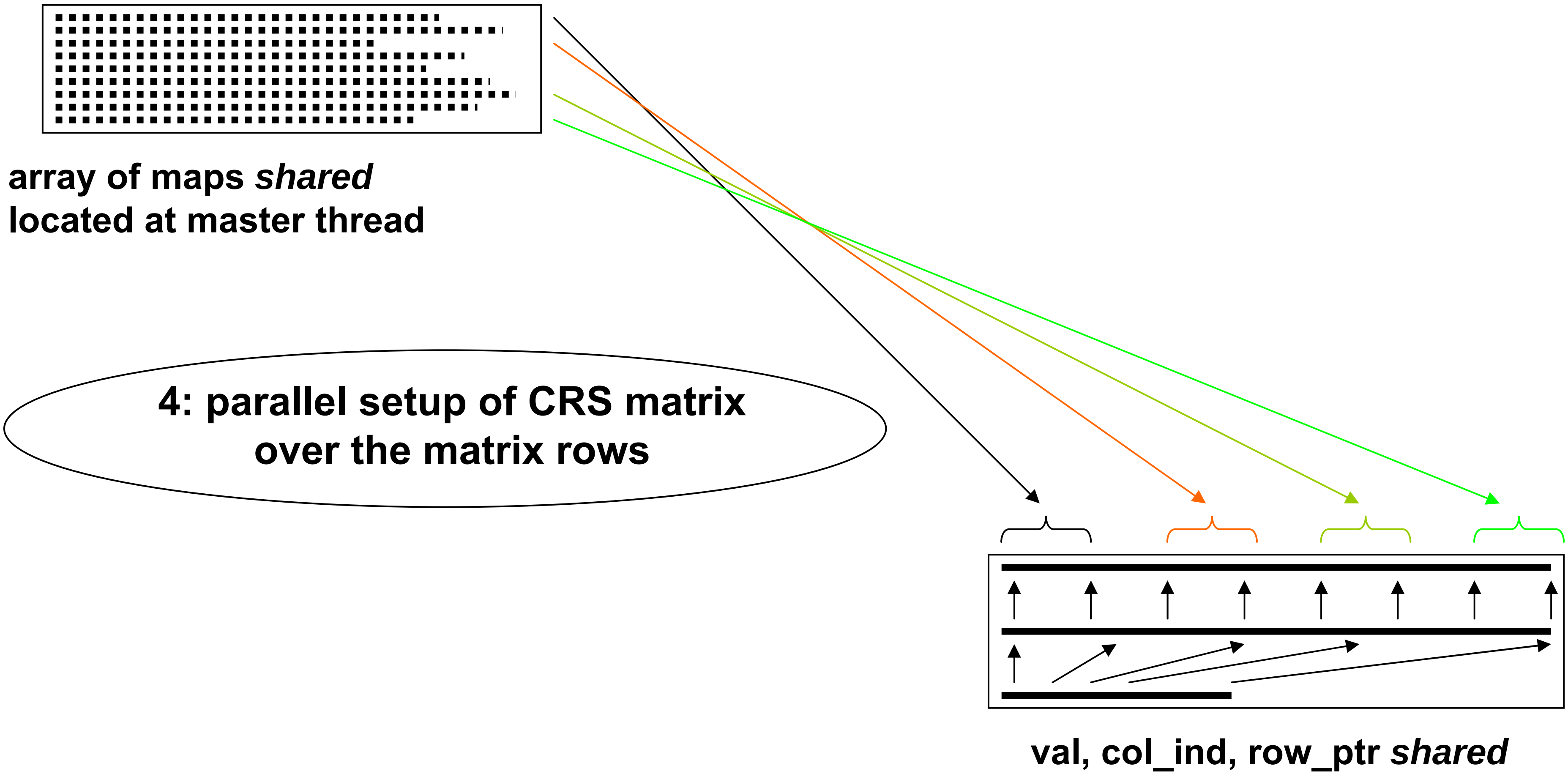
The OpenMP Approach
Assembly of the Stiffness Matrices



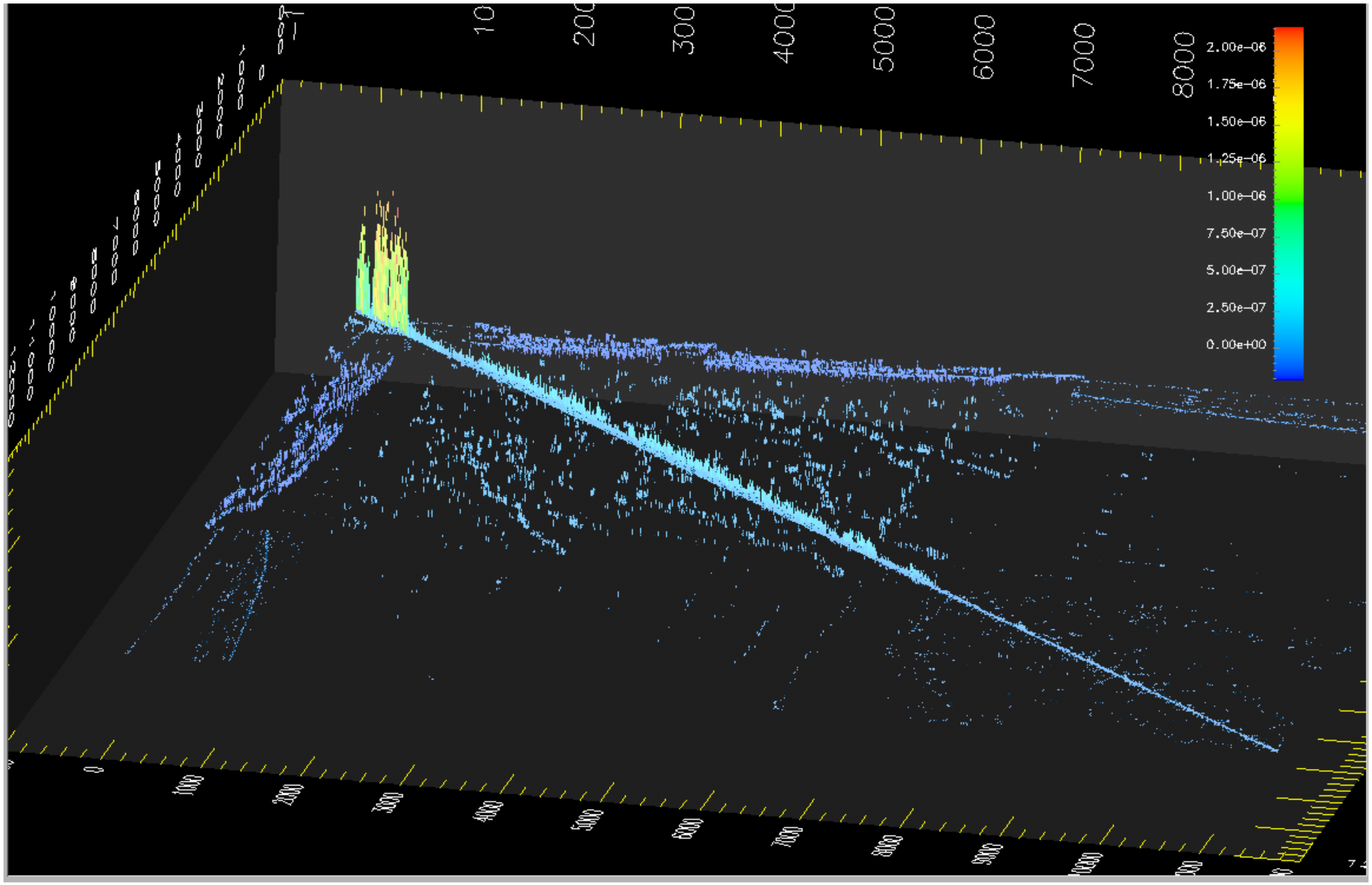
3: master only: setup of CRS structure

val, col_ind, row_ptr *shared*

The OpenMP Approach
Assembly of the Stiffness Matrices



The OpenMP Approach
The Linear Equation Solvers



The OpenMP Approach
The Linear Equation Solvers

- Given method: **SSOR0 = symmetric successive-over-relaxation with start-vector 0 preconditioner:** $x_i = \frac{\omega}{a_{ii}} \cdot (b_i - \sum_{j=0}^{i-1} a_{ij} \cdot x_j)$

$x_0 = w/a_{00} \cdot b_0$

$x_1 = w/a_{11} \cdot (b_1 - a_{10} \cdot x_0)$

$x_2 = w/a_{22} \cdot (b_2 - a_{20} \cdot x_0 - a_{21} \cdot x_1)$

$x_3 = w/a_{33} \cdot (b_3 - a_{30} \cdot x_0 - a_{31} \cdot x_1 - a_{32} \cdot x_2)$

$x_4 = w/a_{44} \cdot (b_4 - a_{40} \cdot x_0 - a_{41} \cdot x_1 - a_{42} \cdot x_2 - a_{43} \cdot x_3)$

$x_5 = w/a_{55} \cdot (b_5 - a_{50} \cdot x_0 - a_{51} \cdot x_1 - a_{52} \cdot x_2 - a_{53} \cdot x_3 - a_{54} \cdot x_4)$

→ denote the critical path

Computation is done **block-wise** parallel with respect to the dependencies of the critical path.

Blocksize: 128

The OpenMP Approach
The Linear Equation Solvers

- Experiment**: JAC0 = Jacobi method with start-vector 0:**

$$x_i = \frac{b_i}{a_{ii}}$$

- Leads to an **increased number of iterations** in each step, but the **total runtime is reduced**: matrix-vector-multiply is more efficient.

- TODO: use a better parallel preconditioner

The OpenMP Approach
The Linear Equation Solvers

```
PCG(const Mat& A, Vec& x, const Vec& b,
    const PreCon& M, int& max_iter,
    double& tol)
{
  Vec 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;
    […]
```

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

```
#pragma omp for reduction
                (+:alpha_sum)
for (long j=0; j<n; j++)
   alpha_sum += p[j]*q[j];

#pragma omp single {
   alpha = rho/alpha_sum;
}
```

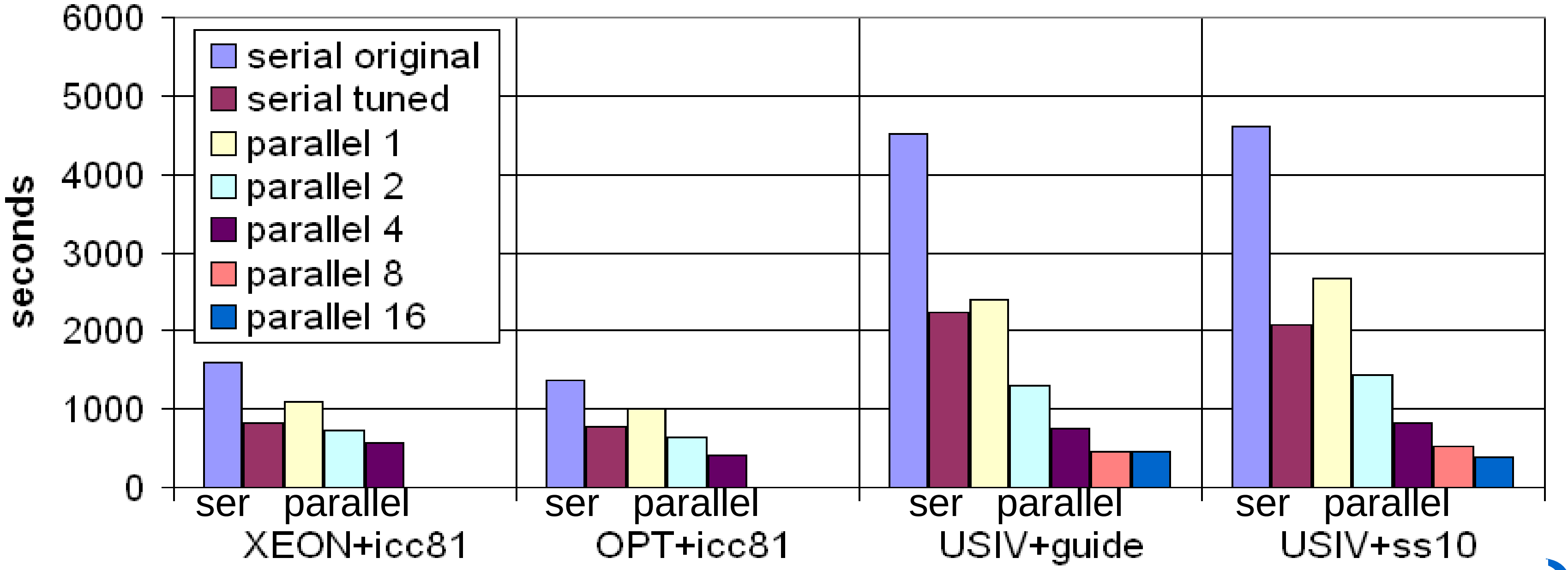
```
#pragma omp for
for (long j=0; j<n; j++){
   x[j] += alpha * p[j];
   r[j] -= alpha * q[j];
}
```

The OpenMP Approach
Compilers

code	DROPS serial	OpenMP support	DROPS parallel
XEON+gcc333	ok	no	n.a.
XEON+gcc343	ok	no	n.a.
XEON+icc81	(ok)	yes	ok
XEON+pgi60	(ok)	yes	compilation fails
XEON+vs2005	compilation fails	yes	compilation fails
OPT+gcc333	ok	no	n.a.
OPT+gcc333X	ok	no	n.a.
OPT+icc81	(ok)	yes	ok
OPT+icc81X	(ok)	yes	compilation fails
OPT+pgi60	(ok)	yes	compilation fails
OPT+pgi60X	(ok)	yes	compilation fails
OPT+path20	ok	no	n.a.
OPT+path20X	ok	no	n.a.
OPT+ss10	(ok)	yes	compilation fails
USIV+gcc331	ok	no	n.a.
USIV+ss10	(ok)	yes	ok
USIV+guide	(ok)	yes	ok
POW4+guide	(ok)	yes	ok
POW4+xlC60	compilation fails	yes	compilation fails
POW4+gcc343	ok	no	n.a.
IT2+icc81	(ok)	yes	1 thread only

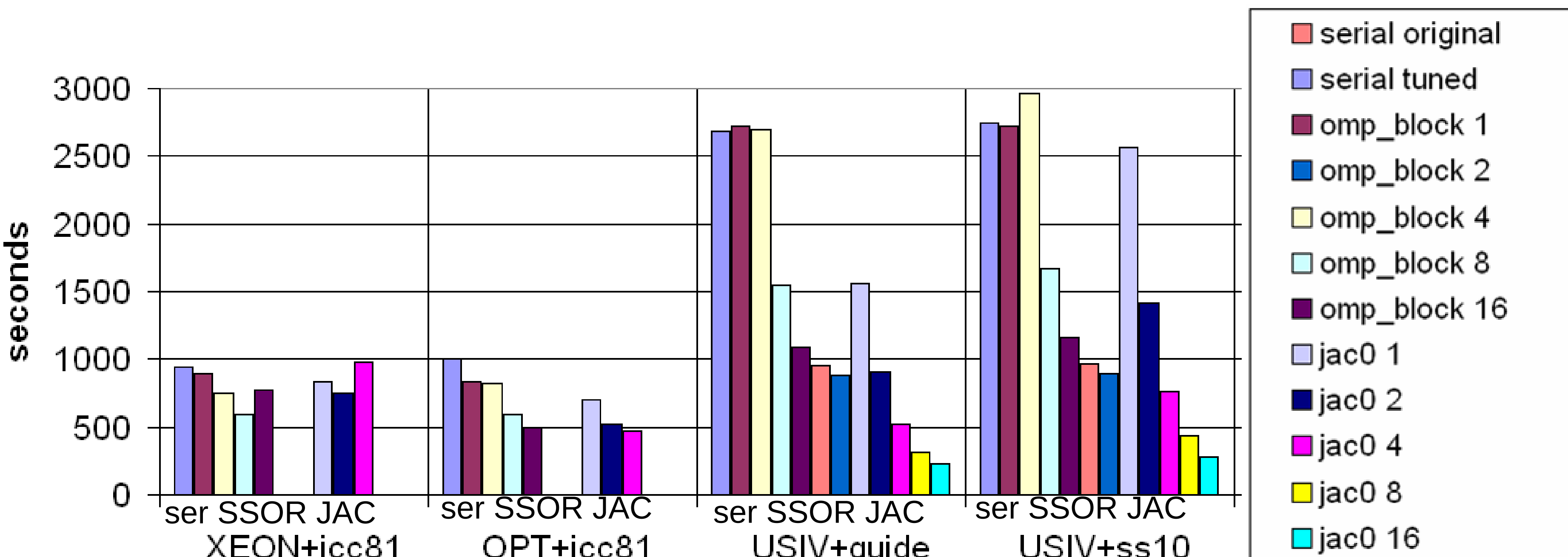
Performance of the OpenMP Version
Assembly of the Stiffness Matrices

code	serial original	serial tuned	parallel				
			1	2	4	8	16
XEON+icc81	1592	816	1106	733	577		
OPT+icc81	1368	778	1007	633	406		
USIV+guide	4512	2246	2389	1308	745	450	460
USIV+ss10	4604	2081	2658	1445	820	523	383



Performance of the OpenMP Version
Linear Equation Solvers

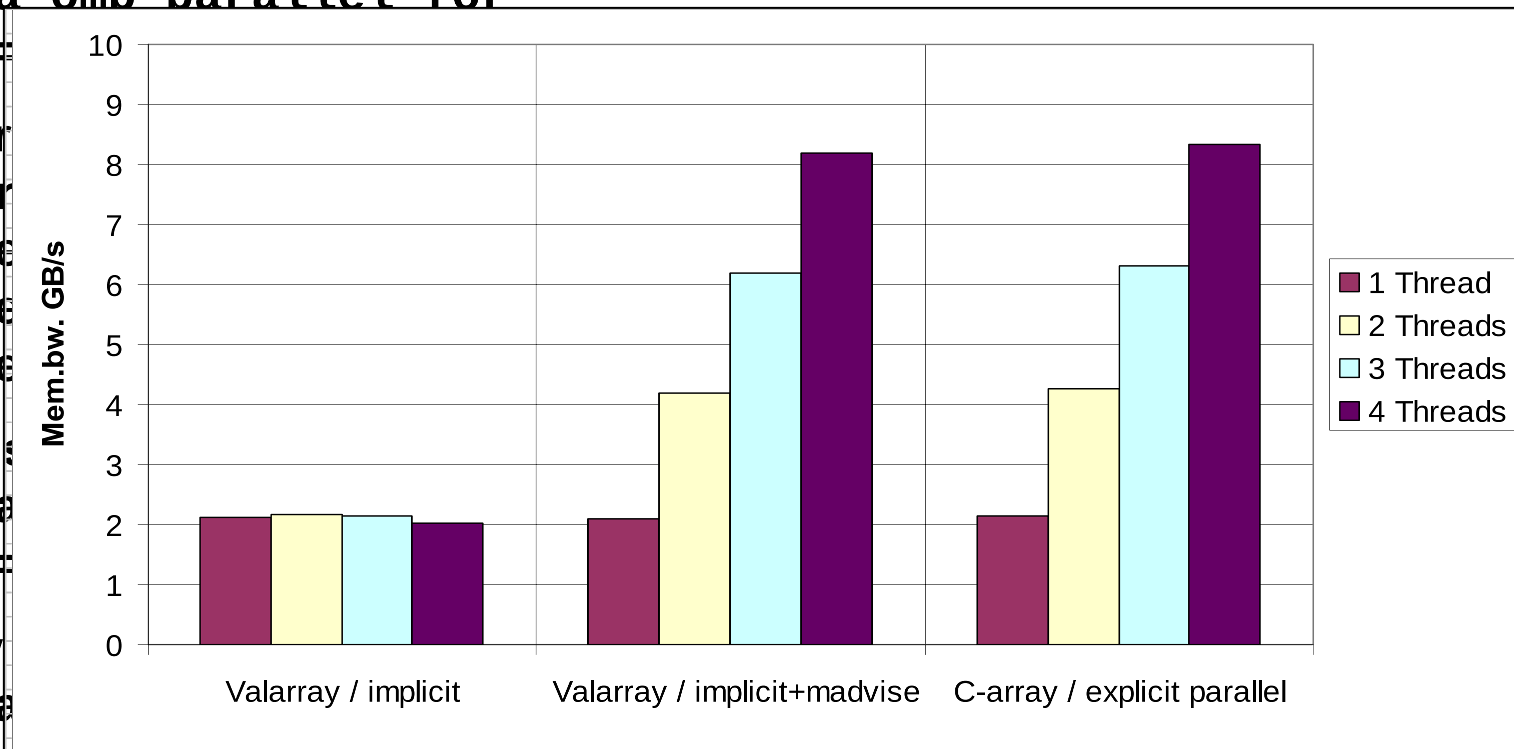
code	serial original	serial tuned	parallel (SSOR)					parallel (JAC)				
			1	2	4	8	16	1	2	4	8	16
XEON+icc81	939	894	746	593	780			837	750	975		
OPT+icc81	1007	839	823	590	496			699	526	466		
USIV+guide	2682	2727	2702	1553	1091	957	878	1563	902	524	320	232
USIV+ss10	2741	2724	2968	1672	1162	964	898	2567	1411	759	435	281



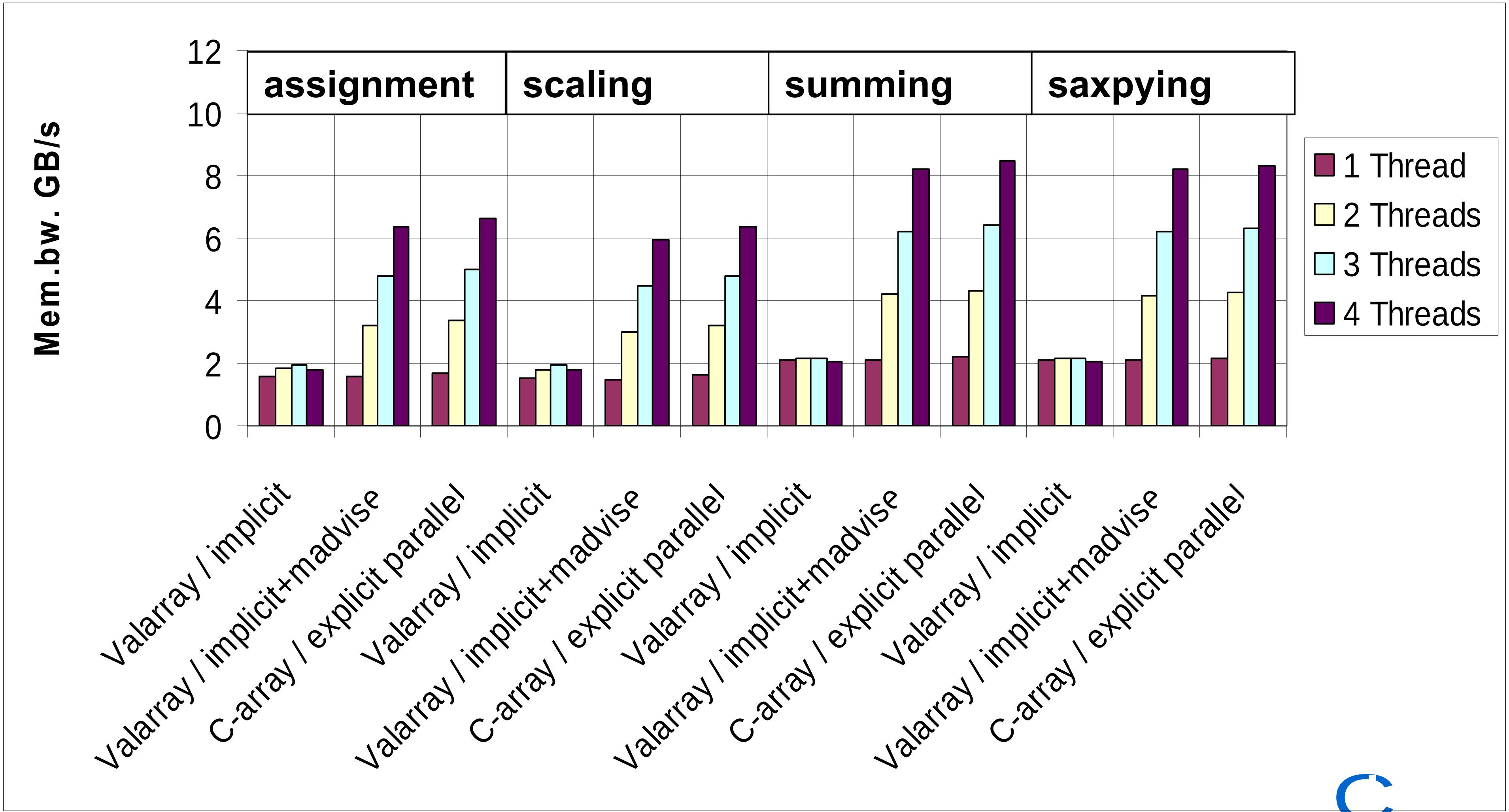
Performance of the OpenMP Version Stream Benchmark (saxpying)

```
//C arrays
doubl
a=(d
b=(d
c=(d

//fir
# pr
for

//sa
#pra
for(
```

```
// C++ valarray STL containers are initialized
// automatically and allocated on the master's memory
valarray<double> a(N), b(N), c(N);

// saxpying is slow
#pragma omp parallel for
for (i=

// migr
// # pr
madvise
madvise
madvise

// still s
#pragma
for (i=

// now
#pragma
for (i=0;i<N;i++) a[i]=b[i]+scalar*c[i];
```

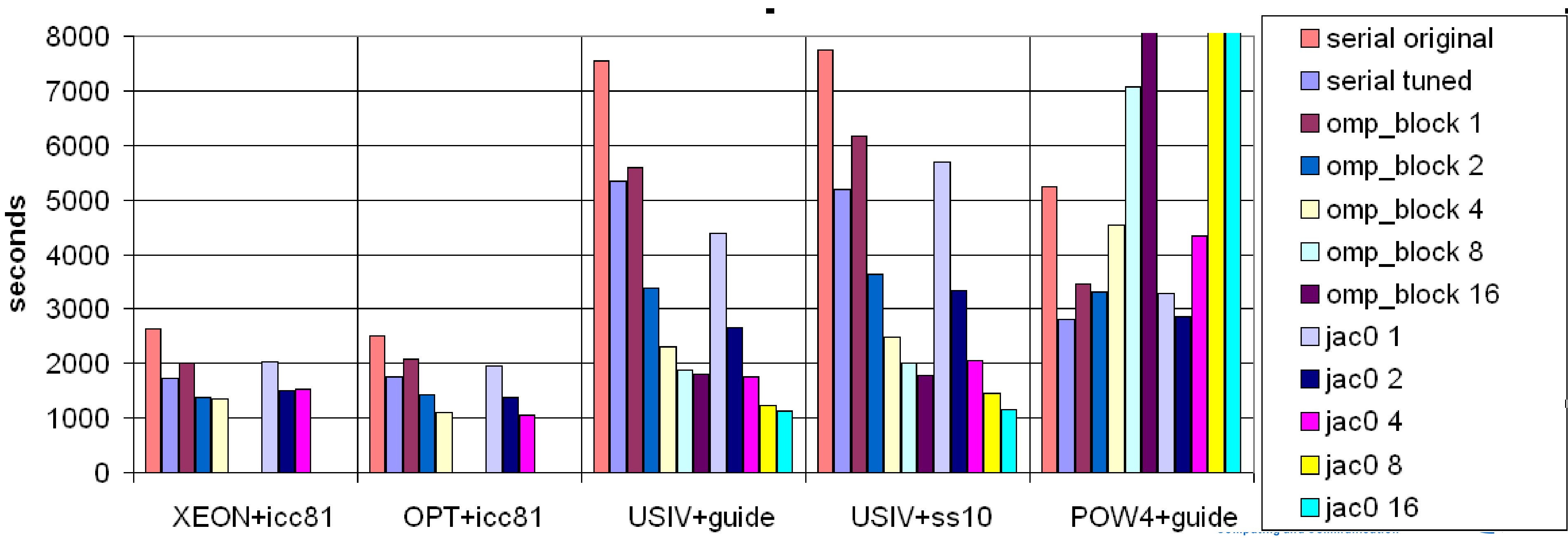


Performance of the OpenMP Version Stream Benchmark

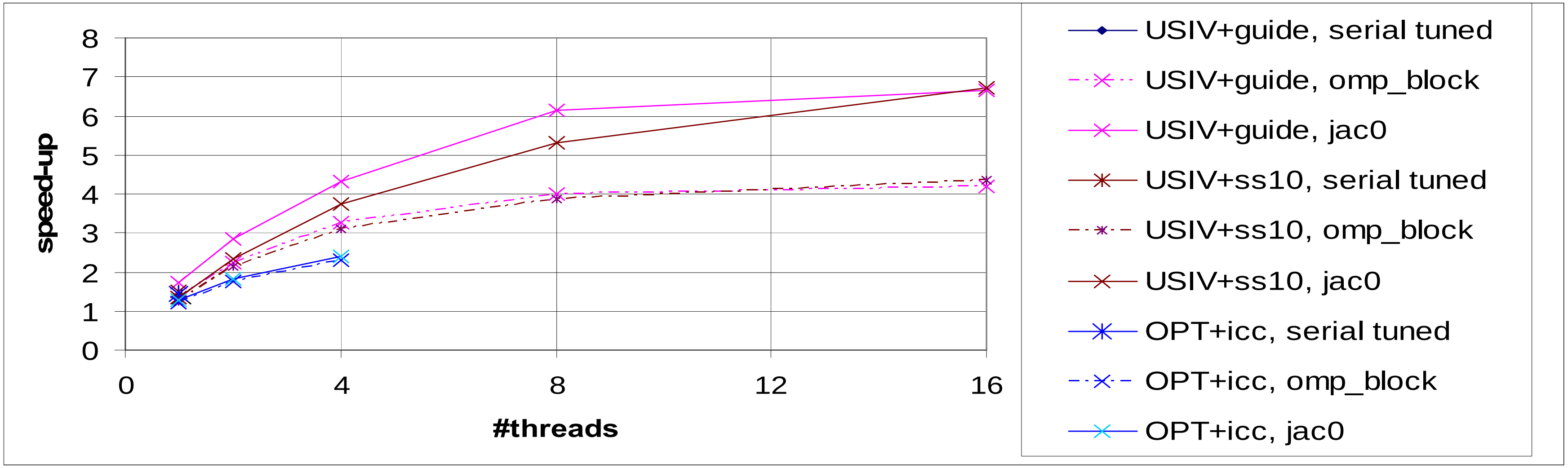


Performance of the OpenMP Version
Total Runtime

code	serial original	serial tuned	parallel (omp_block)					parallel (jac0)				
			1	2	4	8	16	1	2	4	8	16
XEON+icc81	2643	1723	2001	1374	1353			2022	1511	1539		
OPT+icc81	2517	1761	2081	1431	1093			1962	1382	1048		
USIV+guide	7551	5335	5598	3374	2319	1890	1796	4389	2659	1746	1229	1134
USIV+ss10	7750	5198	6177	3629	2488	2001	1782	5683	3324	2067	1457	1151



Performance of the OpenMP Version
Total Speed-up



Version	USIV+guide				USIV+ss10				OPT+icc			
	omp_block		jac0		omp_block		jac0		omp_block		jac0	
serial(original)	1.00		1.00		1.00		1.00		1.00		1.00	
serial(tuned)	1.42				1.49				1.43			
parallel(1 thread)	1.35	1.00	1.72	1.00	1.26	1.00	1.36	1.00	1.21	1.00	1.28	1.00
parallel(2 threads)	2.24	1.66	2.84	1.65	2.14	1.70	2.33	1.71	1.76	1.45	1.82	1.42
parallel(4 threads)	3.26	2.41	4.32	2.51	3.11	2.47	3.75	2.76	2.30	1.90	2.40	1.88
parallel(8 threads)	3.99	2.96	6.14	3.57	3.87	3.07	5.32	3.91				
parallel(16 threads)	4.20	3.11	6.66	3.87	4.35	3.45	6.73	4.95				

1. The DROPS multi-phase Navier-Stokes solver
2. Portability and Performance of the Serial Program Version
3. The OpenMP Approach
4. Performance of the OpenMP Version
5. **Summary**

Summary
OpenMP + C++

- **C++** approach is very meaningful and elegant for the experimenting with numerical methods. But: **conformance** to standards and **reliability** of compilers is a critical issue.
- Compilers:
  - g++ delivers good performance, but no OpenMP-support yet
  - Intel C++ is compiler of choice on XEON and Opteron
  - Sun C++ 10 is finally compiling the code on USIV
- Iterator loops -> “proper” for-loops in SETUP routines.
- Step back to C coding style in solvers to reduce OpenMP overhead.

Summary OpenMP Parallelization

- **Serial version** could be improved by a factor of ~1.5
- **Parallel speedup up to 5** with 16 threads on UltraSPARC/Solaris
- **Faster processors** (Xeon, Opteron) only in smaller SMP systems
- **ccNUMA** on Opteron: locality of data is necessary, STL constructors initialize data locally on the master's memory.
  => need for a **NEXTTOUCH** OpenMP directive?

- Additional **parallel overhead** in matrix setup.
- **Simplified preconditioner** (jac0) scales nicely, but causes higher iteration count (more work to be done by the IGPM).

Thank you for

your attention!

Center for
Computing and Communication