HPCSE II

ScaLAPACK and TBB

Libraries for hybrid and multithreaded programming

PLASMA

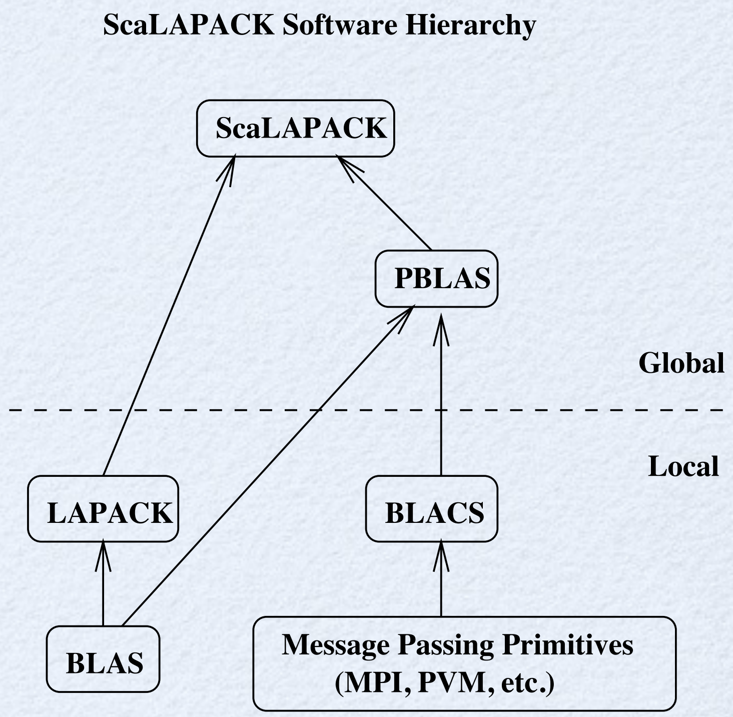
- Parallel Linear Algebra Software for Multicore Architectures
- Multi-threaded rewrite of LAPACK and BLAS functionality
- Optimizes for tiling/blocking and cache reuse on NUMA architectures
- Available from http://icl.cs.utk.edu/plasma

MAGMA

- Matrix Algebra on GPU and Multicore Architectures
- Hybrid CPU/GPU rewrite of LAPACK and BLAS functionality
- Available from http://icl.cs.utk.edu/magma

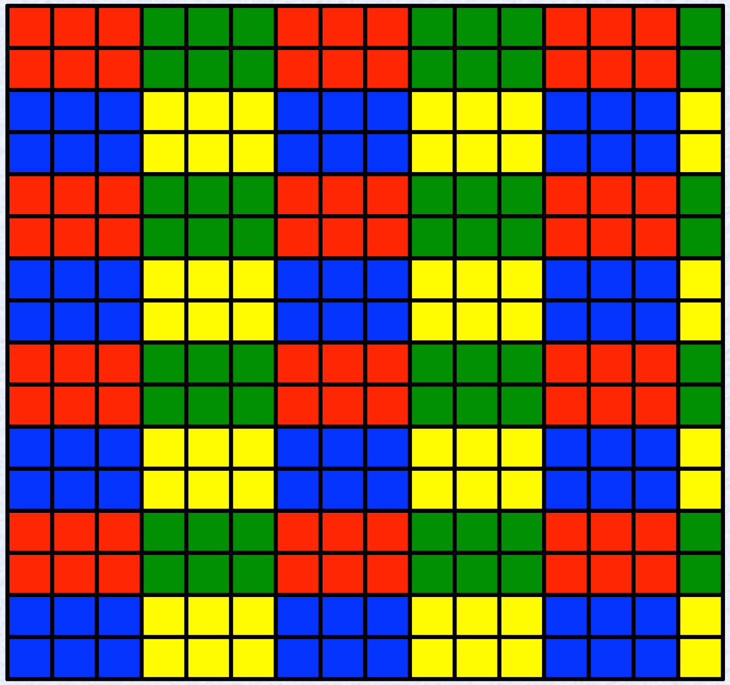
ScaLAPACK

- is a distributed memory extension of BLAS and LAPACK
- available from http://www.netlib.org/scalapack
- Builds upon
  - BLACS for communication abstraction over MPI, PVM or shared memory
  - BLAS and LAPACK for local computations
  - PBLAS a distributed BLAS using block-cyclic distributions


ScaLAPACK Software Hierarchy
ScaLAPACK
PBLAS
Global
Local
LAPACK
BLACS
BLAS
Message Passing Primitives (MPI, PVM, etc.)

Recall: block-cyclic distribution

- Block cyclic distribution
  - Example: 3x2 blocks, 2x2 process array
- Ideal block-sizes machine dependent
  - 32x32 or 64x64 are good starting guesses



Initializing the BLACS layer

- We do not need to initialize MPI ourselves but use the BLACS which typically builds on top of MPI

```
int main(int argc, char** argv)
{
  int rank;
  int nprocs;

  // initialize MPI using BLACS. Using MPI this is similar to the three statements below
  // MPI_Init(&argc, &argv);
  // MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  // MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  Cblacs_pinfo(&rank,&nprocs);

  ...

  // we are done: call MPI_Finalize()
  Cblacs_exit(0);

  return 0;
}
```

Defining the process grid

- Next initialize the process grid and find out who I am

```
 // we want to use 2x3 processes
 int nprow=2;
 int npcol=3;

 // get the system context
 int ctxt;
 Cblacs_get(0,0,&ctxt);

 // initialize a 2x3 process grid
 Cblacs_gridinit(&ctxt,"Row-major",nprow,npcol);

 // get my coordinates in the process grid
 int myrow, mycol;
 Cblacs_gridinfo(ctxt,&nprow,&npcol,&myrow,&mycol);

 // continue only if this rank is actually part of the grid
 if (myrow>=0)  {
   std::cout << "Rank " << rank << " has coordinates " << myrow << " " << mycol << "\n";

  ...

}
else
   std::cout << "Rank " << rank << " is not used \n";
```

Block-cyclic storage

- We need to find the size of our local blocks
  - We will solve A*X=B for a matrix A and a vector B, and

```
// we want a 100x 100 matrix
int n=100;

// and we want to solve with 1 right hand side
int nrhs = 1;

// we will use 32x32 size blocks in the block-cyclic layout
int nb=32;

// now intialize the matrix
// first calculate how many rows (np) and columns (nq) we store locally
int np = numroc_(n,nb,myrow,0,nprow);
int nq = numroc_(n,nb,mycol,0,npcol);
int nqrhs = numroc_(nrhs,nb,mycol,0,npcol);

// allocate local storage
hpc12::matrix<double> A(np,nq);
hpc12::matrix<double> B(np,nqrhs);

// create descriptors for the matrix and right hand side
int descA[9], descB[9], info;
descinit_( descA, n, n   , nb, nb, 0, 0, ctxt, A.leading_dimension(), &info );
descinit_( descB, n, nrhs, nb, nb, 0, 0, ctxt, B.leading_dimension(), &info );
```

Finally call the solvers

- Solve the equation using **pdgesv**
- then use **pdgemm** to calculate the residual
- and finally use **pdlange** to get its 1-norm
  - pdlange needs work space. The size is taken from the manual.

```
    // call the parallel solver
    std::vector<int> pivot(np+nb);
    pdgesv_(n, nrhs, A.data(), 1, 1, descA, &pivot[0], X.data(), 1, 1, descB, &info );
    assert(info==0);

    // now call pdgemm to calculate the 1-norm of the residual ||A * X  - B||_1
    pdgemm_( "N", "N", n, nrhs, n, 1., Acopy.data(), 1, 1, descA,
                                       X.data(),     1, 1, descB,
                                  -1., B.data(),     1, 1, descB);

    int workspace   = numroc_( n, nb, mycol, indxg2p_(1, nb, mycol, 0, npcol ), npcol );
    std::vector<double> work(workspace);
    double rnorm = pdlange_( "1", n, nrhs, B.data(),     1, 1, descB, &work[0]);

    if ( rank==0 )
        std::cout <<  "1 norm of residual: " << rnorm << "\n";
```

Intel TBB

Intel Thread Building Blocks

- A threading library by Intel, based on a draft version of C++11
  - implements C++11 threads plus a few extra locks and mutexes
  - higher level algorithmic abstractions
    - parallel_for
    - parallel_do
    - parallel_while
    - tasks
  - thread-local data
  - thread-safe data structures

Parallel for

- Consider a simple serial loop:

```
int main() {
  const int n = 10000;
  std::vector<double> x(n);

  for (int i=0; i<n ; ++i)
    x[i]=std::sin(i*0.01);

}
```

- We want to run it in parallel using tbb::parallel_for

```
#include <tbb/parallel_for.h>
#include <tbb/blocked_range.h>

int main() {

  const int n = 10000;
  std::vector<double> x(n);

  tbb::parallel_for( tbb::blocked_range<int>(0, n),
    [&] (tbb::blocked_range<int> const& r) {
      for( int i=r.begin(); i!=r.end(); ++i )
         x[i]=std::sin(i*0.01);
    });
}
```

Parallel for (continued)

- The parallel_for

```
tbb::parallel_for( tbb::blocked_range<int>(0, n), ...);
```

- needs a parallel function object that takes a blocked_range. We used a C++11 lambda function

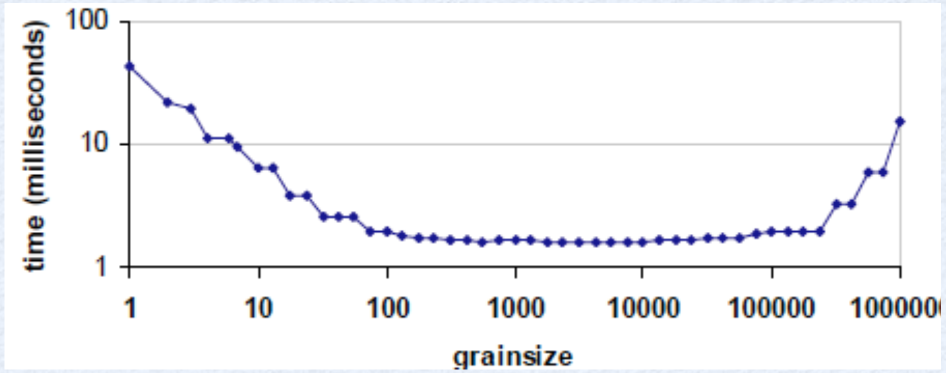
```
[&] (tbb::blocked_range<int> const& r) {
      for( int i=r.begin(); i!=r.end(); ++i )
         x[i]=std::sin(i*0.01);
    }
```

- TBB automatically launches threads and splits the range

Controlling chunking

- tbb::parallel_for will be profitable when the loop runs for about 500µs
- Further optimization can be obtained by controlling the chunking:

```
tbb::parallel_for(tbb::blocked_range<int>(0,n,G), f, tbb::simple_partitioner());
```



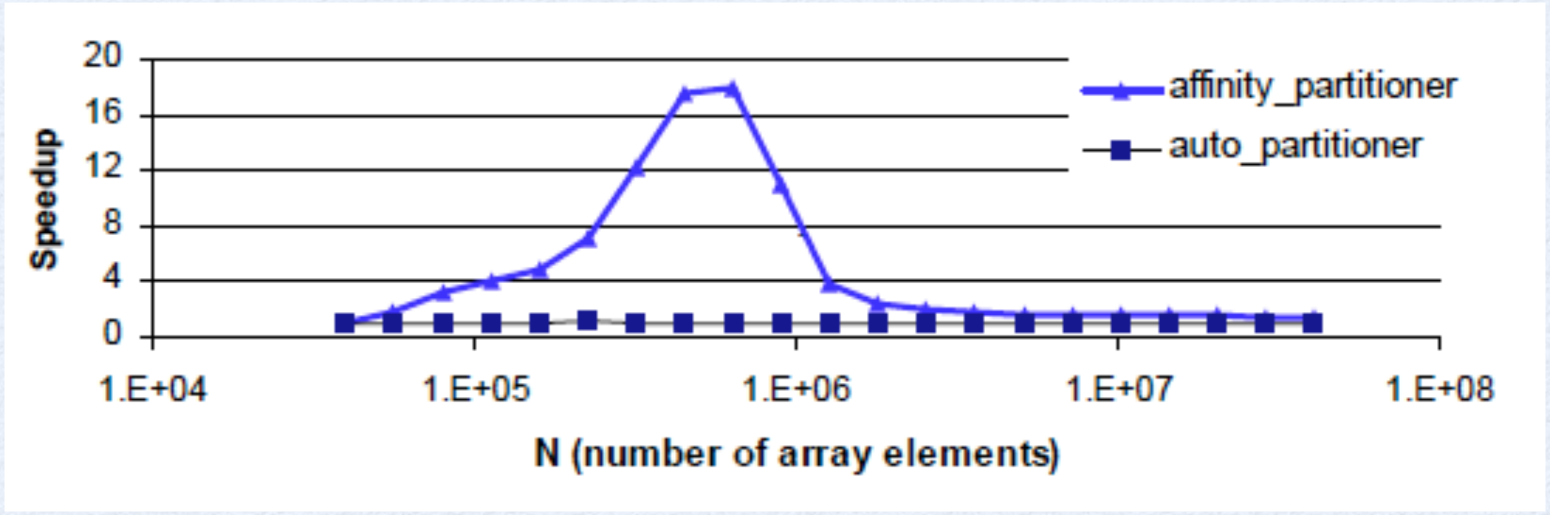
- The **simple_partitioner** splits the loop into chunk sizes between G/2 and G
- The **auto_partitioner** is the default
- The **affinity_partitioner** can improve cache locality

The affinity partitioner

- When running a loop many times the affinity partitioner can improve cache affinity. The partitioner remembers which thread did which chunk

```
tbb::affinity_partitioner ap; // create a partitioner that is reused
for (int i=0 ; i< iterations; ++i)
  tbb::parallel_for(tbb::blocked_range<int>(0,n), f, ap);
```

- Intel shows substantial speedup is possible:



Parallel reductions

- A reduction

```
double sum=0.;
for (int i=0; i<n ; ++i)
  sum += x[i];
```

- can be done by parallel_reduce

```
sumit s(x);
tbb::parallel_reduce( tbb::blocked_range<int>(0, n), s);
```

- but now it needs a more complex function object and not just a lambda function

Parallel reduce (continued)

```
class sumit {
public:
  sumit(std::vector<double> const&  x)
  : data_(x)
  , sum_(0.)
  {}

  sumit(sumit const& x, tbb::split )
  : data_(x.data_)
  , sum_(0.)
  {}

  void join(sumit const& y ) {sum_ += y.sum_;}

  void operator()(tbb::blocked_range<int> const& r )
  {
    double s = sum_; // remember to add to partial sum done locally
    for( int i=r.begin(); i!=r.end(); ++i )
      s += data_[i];
    sum_ = s;
  }

  double sum() const { return sum_;}

private:
  double sum_;
  std::vector<double> const& data_;
};
```

- The splitting constructor is used to make copies for other threads that should copy the data but not the partial sums

Thread-local data in TBB: combinable

```
 template <typename T>
class combinable {
  public:
    combinable();
    template <typename FInit>
    combinable(FInit finit);                    // constructed for each thread from return value
    ...

    void clear();


    T& local();
    T& local(bool & exists);


    template<typename FCombine>
    T combine(FCombine fcombine); // a reduction over all threads


    template<typename Func>
    void combine_each(Func f);        // a function applied to each thread’s value
};
```

The reduction using combinable

- Use thread-local storage and combine it at the end

```cpp
tbb::combinable<double> sum(0.);
tbb::parallel_for( tbb::blocked_range<int>(0, n),
  [&] (tbb::blocked_range<int> const& r) {
    double s=0.;
    for( int i=r.begin(); i!=r.end(); ++i )
      s +=x[i];
    sum.local() += s;
  });

std::cout << "The sum is " << sum.combine(std::plus<double>()) << "\n";
```

Thread-local data in TBB: enumerable_thread_specific

- is a container with one element per thread. Easiest to see in use, again for the reduction

```
tbb::enumerable_thread_specific<double> sum(0.);
tbb::parallel_for( tbb::blocked_range<int>(0, n),
  [&] (tbb::blocked_range<int> const& r) {
    double s=0.;
    for( int i=r.begin(); i!=r.end(); ++i )
      s +=x[i];
    sum.local() += s;
  });

std::cout << "The sum is " << std::accumulate(sum.begin(),sum.end(),0.) << "\n";
```

More parallel operations

- TBB contains further and more flexible parallelization constructs:
  - Arbitrary iteration spaces
  - **parallel_do**
    - Iteration through lists
    - Iteration through trees, offloading children to other threads
  - **parallel_pipeline**
    - for pipelining various steps done on a continuous stream of input data
- When should one use TBB?
  - If you need finer control than OpenMP but don’t want to manage the threads yourself manually
- Is it worth it?
  - That depends on whether you prefer to read manuals or code yourself.

Thread safe containers

- Thread safe containers circumvent the need to always lock a data structure manually for parallel access. TBB contains
  - **concurrent_hash_map**
  - **concurrent_vector**
  - **concurrent_queue** and **concurrent_bounded_queue**

Example: concurrent_vector

- is similar to std::vector, but
  - **might not be contiguous in memory**
  - allows concurrent insertion by
    - v.push_back(x);
    - v.grow_by(n);
    - v.grow_to_at_least(n);
  - allows concurrent iteration and check for size()

- **However, watch out**:
  - only iteration and insertion is thread-safe
  - access to an element still needs to be synchronized manually
  - calling clear() is not thread-safe

concurrent_vector example

- All threads can push safely into the same vector

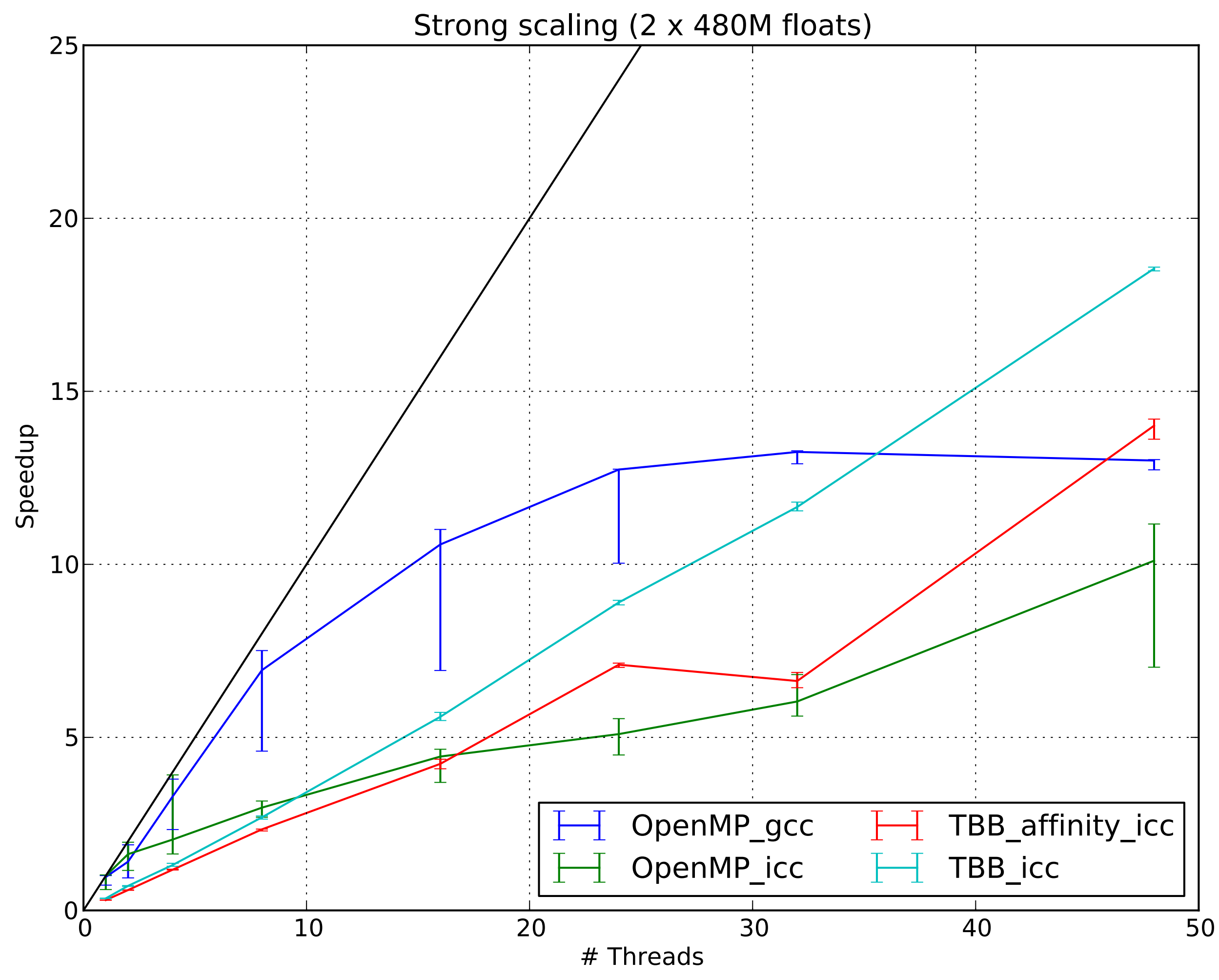
```
#include <tbb/concurrent_vector.h>
#include <tbb/blocked_range.h>
#include <tbb/parallel_for.h>
#include <random>
#include <iostream>


int main() {

  tbb::concurrent_vector<double> x;
  const int n = 100000;

  tbb::parallel_for( tbb::blocked_range<int>(0, n),
    [&] (tbb::blocked_range<int> const& r) {
      std::mt19937 mt;
      std::uniform_real_distribution<double> ureal_d(0.,10.);
      for( int j = r.begin(); j !=r.end(); ++j )
        x.push_back(ureal_d(mt)); // concurrent push_back is safe!
      });

  std::cout << " The size is " << x.size() << "\n";
  return 0;
}
```

Task scheduler

- TBB provides **tasks** as an abstraction over thread
  - split a job into many tasks
  - TBB will schedule the tasks over the available threads
  - creating a task is 10-100 times faster than creating a thread
- How does it compare to OpenMP tasks?
  - Much harder to create and manage
  - But finer control over dependencies
- Is it worth it?

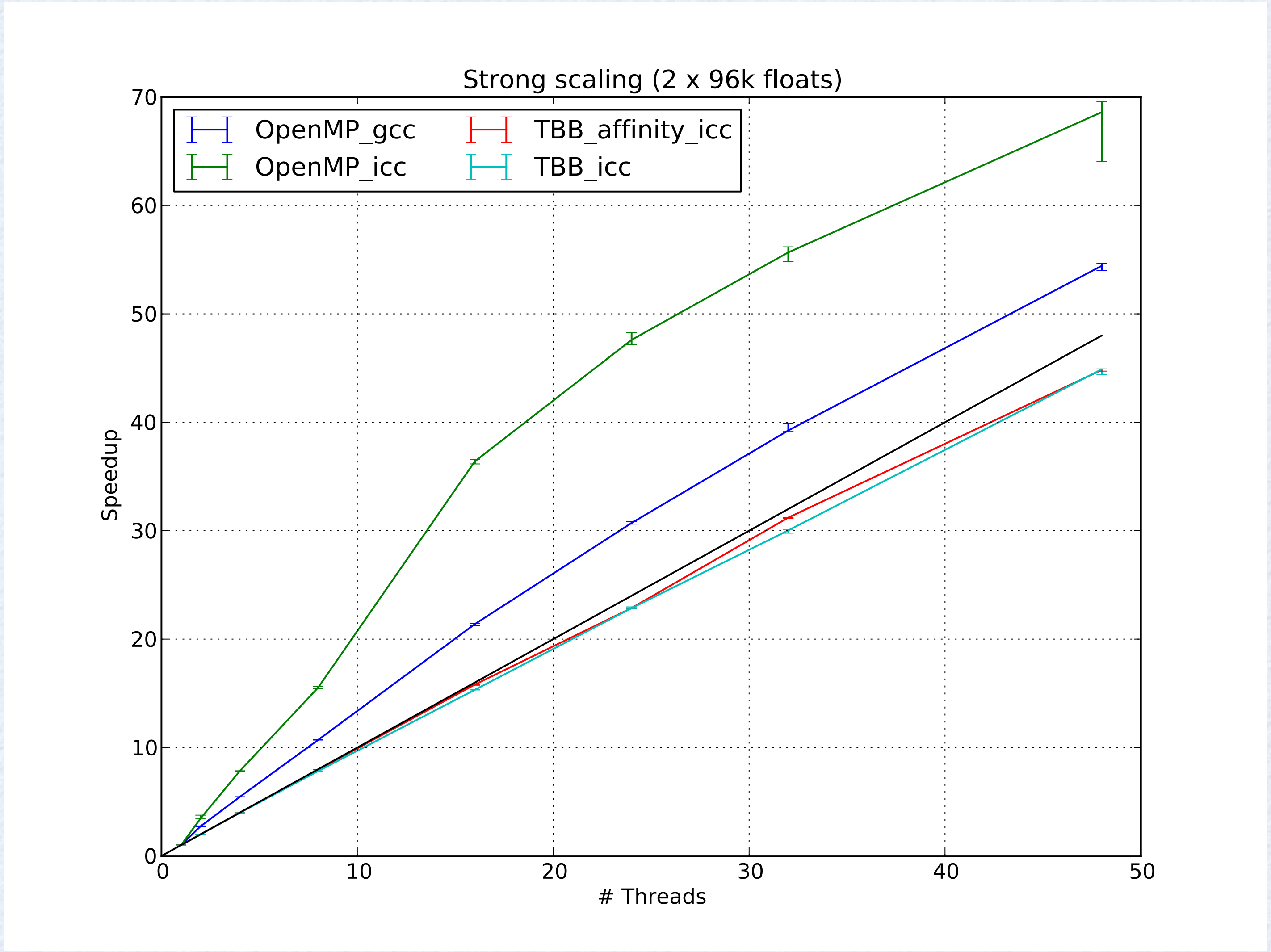
Some benchmarks by Andreas Hehn

- Multi-threaded vector multiplications, taking care of NUMA effects
- Lesson learned: benchmarks are the only way to tell what is fastest



Some benchmarks by Andreas Hehn

- Multi-threaded vector multiplications, taking care of NUMA effects
- Lesson learned: benchmarks are the only way to tell what is fastest


Strong scaling (2 x 96k floats)
OpenMP_gcc
OpenMP_icc
TBB_affinity_icc
TBB_icc
Speedup
# Threads