Efficient dynamic parallelization for linear algebra algorithms

Software and Services Group, Intel*, Novosibirsk

Alexander Kobotov

Oct 2008

Technical Collateral Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site(www.intel.com).

Agenda

- Introduction to dynamic parallelization
- QR and Cholesky factozations for dynamic parallelism
- Results achieved

Parallelization

Static – Work is distributed on entering a parallel region.

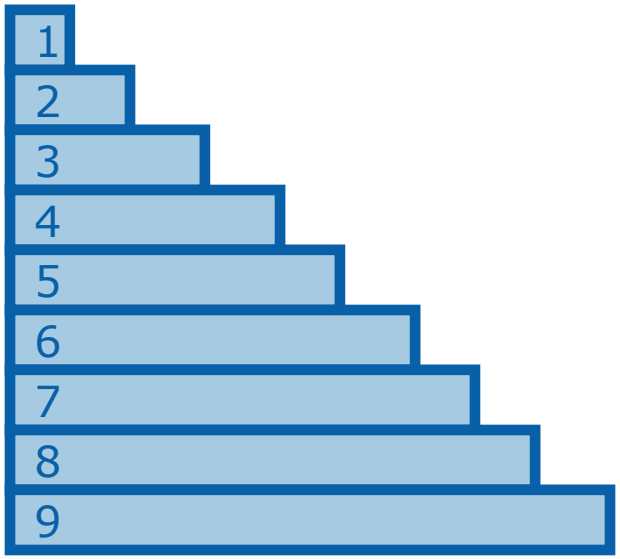
+ Easy to implement
– Bad work balance (It is difficult to load threads equally)

Dynamic – Free thread gets next available piece of work.

– Complex implementation (weak support in OpenMP 2.5, dependency control)
– Synchronization costs
+ Good utilization of multi-core resources.
+ Adjusts with undetermined OS behavior and unknown efficiency of sub algorithms

Dynamic scheduling is not a panacea

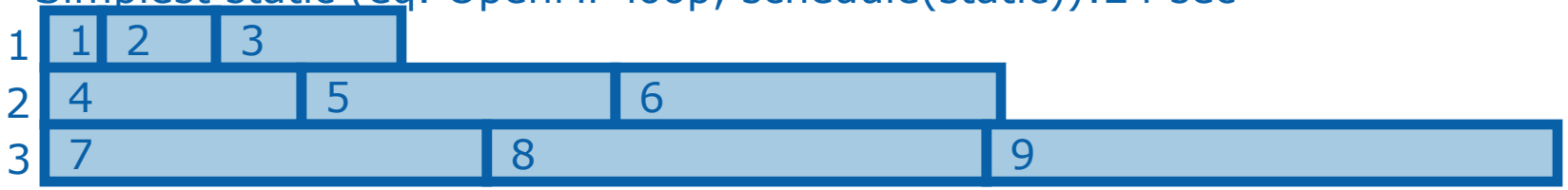
Let’s try to adjust these 9 workloads to 3 threads*:



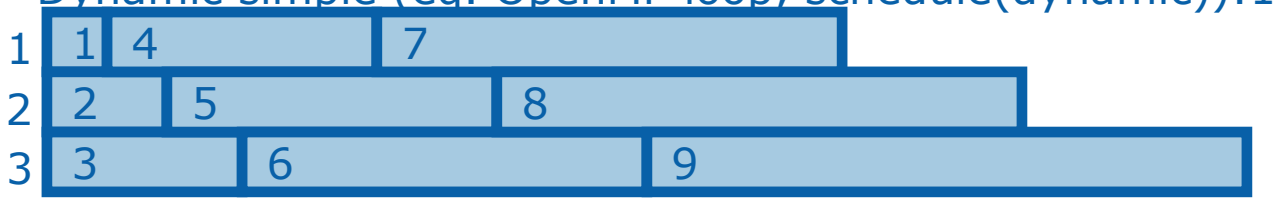
* - this is a common case for symmetric matrices
The number of workload is equal to estimated time in seconds.
Let assume that there are no dependencies between workloads.

Dynamic scheduling is not a panacea

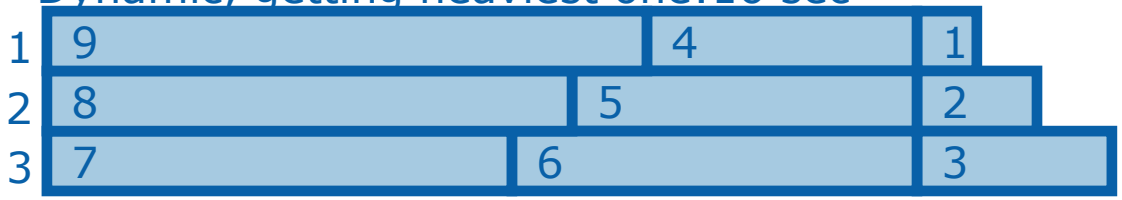
Simplest static (eg. OpenMP loop, schedule(static)):24 sec



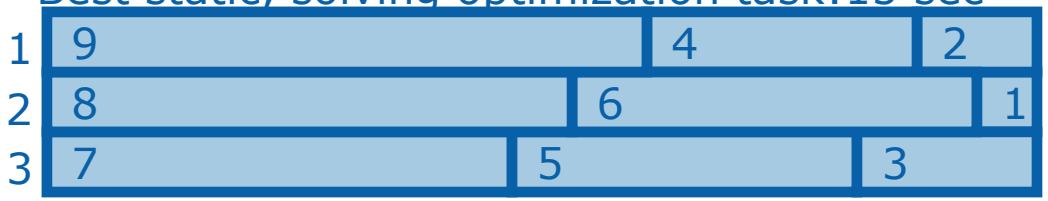
Dynamic simple (eg. OpenMP loop, schedule(dynamic)):18 sec



Dynamic, getting heaviest one:16 sec

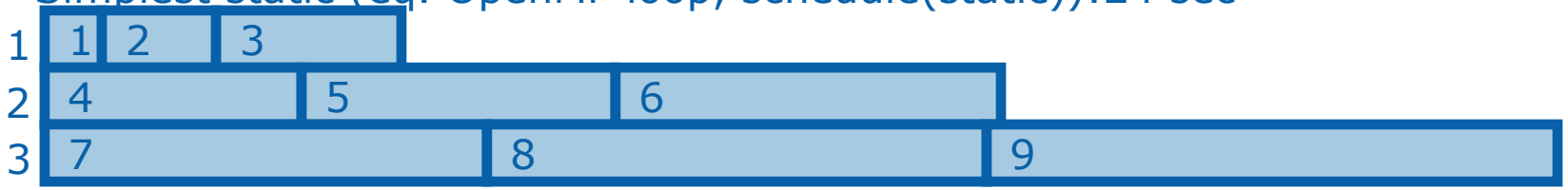


Best static, solving optimization task:15 sec

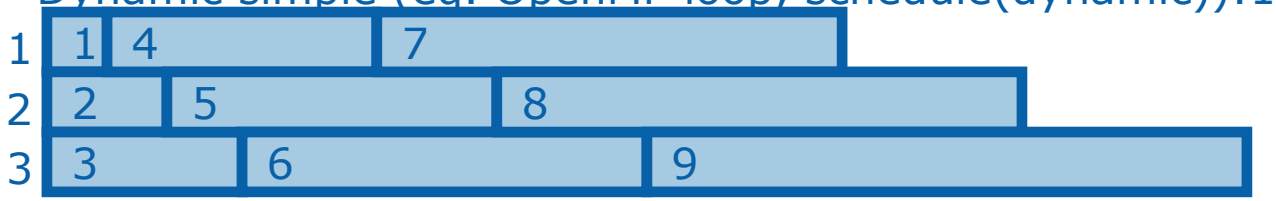


Dynamic scheduling is not a panacea

Simplest static (eg. OpenMP loop, schedule(static)):24 sec

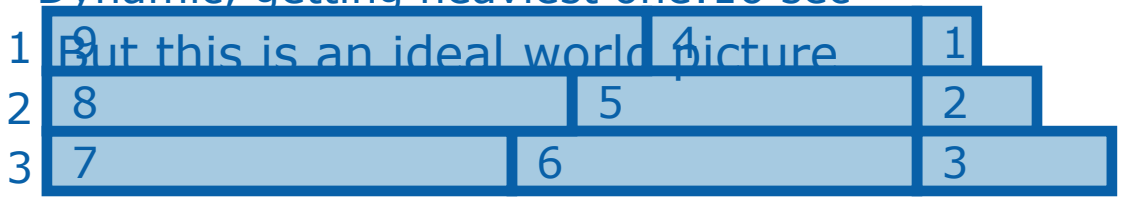


Dynamic simple (eg. OpenMP loop, schedule(dynamic)):18 sec

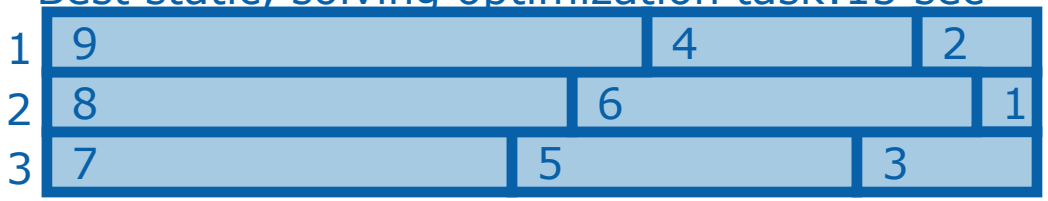


Dynamic, getting heaviest one:16 sec

But this is an ideal world picture

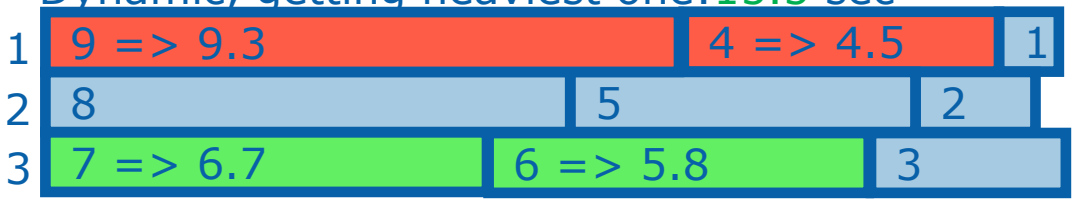


Best static, solving optimization task:15 sec

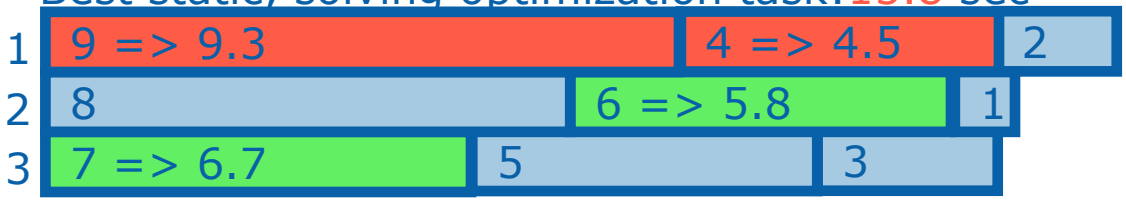


Dynamic scheduling is not a panacea, but looks the best one if smart enough

Dynamic, getting heaviest one: 15.5 sec



Best static, solving optimization task: 15.8 sec



Real workloads are difficult to measure due to:

- Influence of other OS processes
- Data/code cache misses/looses
- Data alignment
- Complicated low level code due to compiler optimizations
- Dependencies from machine configuration
- etc. etc.

A short intro to QR and Cholesky(LL^T) factorizations

QR factorization of matrix ***A*** *mxn*:

$$\boldsymbol{A=QR} \qquad \begin{pmatrix} x & x & x \\ x & x & x \\ x & x & x \end{pmatrix} = Q \begin{pmatrix} x & x & x \\ & x & x \\ & & x \end{pmatrix}$$

Q – orthogonal *mxn* и ***R*** – upper triangular *nxn*.

It's stable and used for:

- Solving of linear systems with bad condition numbers
- Least squares
- In singular decomposition algorithm

FLOPS* numbers: $(2/3)\min(m,n)^2(3\max(m,n)-\min(m,n))$

or $(4/3)n^3$ when $m=n$

* - (FLOPS) Floating Point Operations

A short intro to QR and Cholesky(LL^T) factorizations

Cholesky factorization of symmetric positive define matrix ***A*** *nxn*:

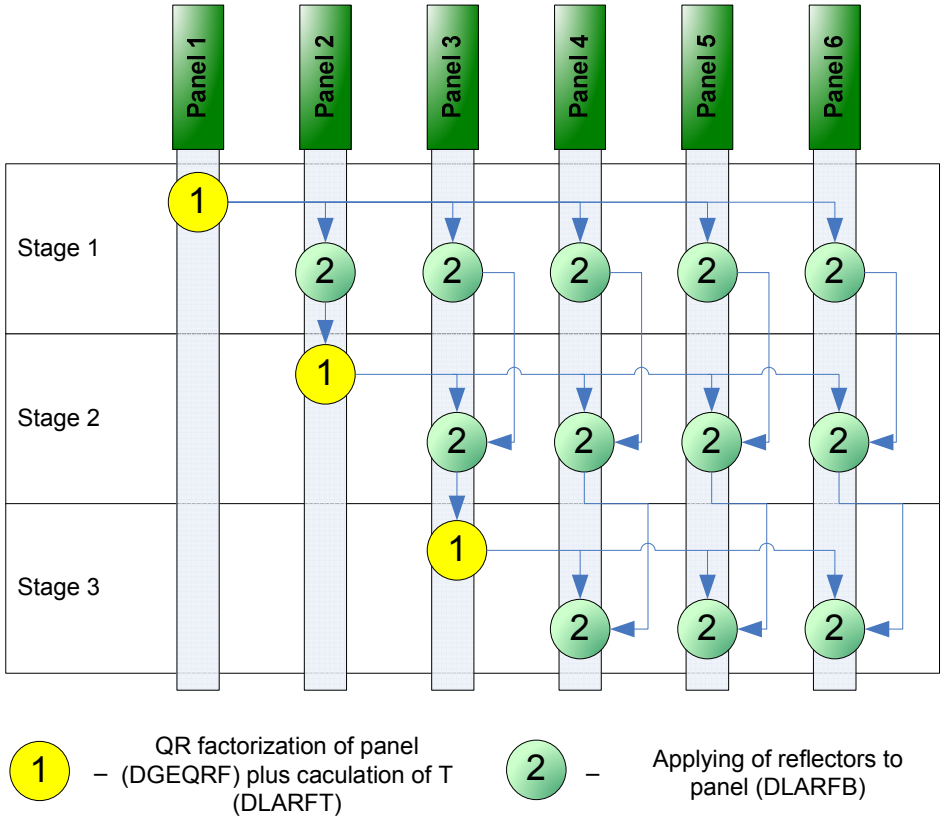
$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T \text{ or } \boldsymbol{A} = \boldsymbol{U}\boldsymbol{U}^T$$

L is lower triangular, ***U*** is upper triangular.

Used for solving of system of linear equation:

- Twice less operations than in LU decomposition
- No permutations
- Only one matrix is on output

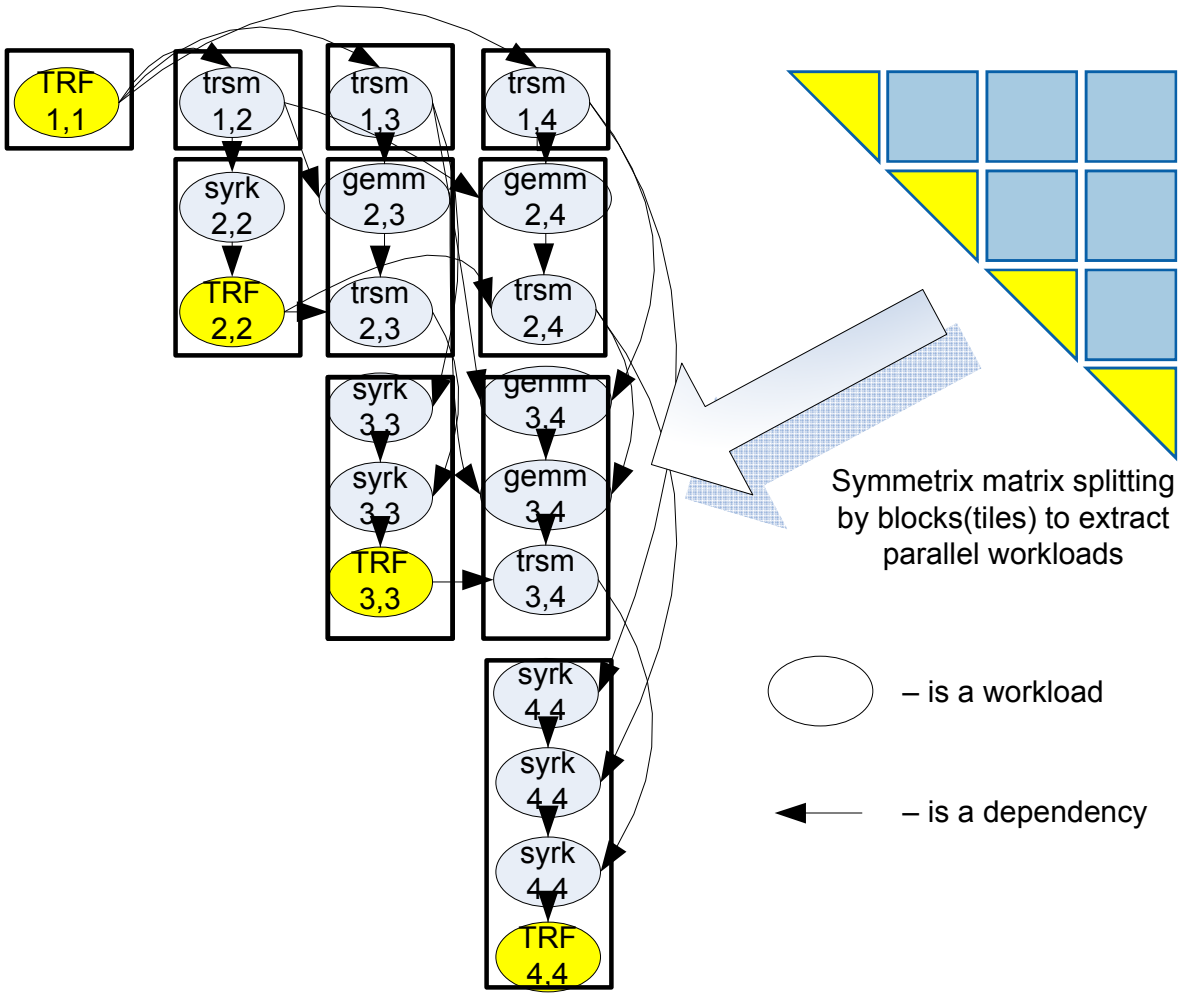
Direct acyclic graph for QR



Matrix is split by panels.

An amount of stages should be applied to every panel taking into account the dependencies shown on graph.

Direct acyclic graph for Cholesky



Subtasks should be effective

To achieve highest performance for each subtask used:

- Merging neighbor subtasks in the beginning of execution – bigger blocks allows higher performance
- Choosing a task from a queue with most recently touched data – cache optimization
- Blocks of data to processor assignment – for NUMA

Graph flow optimizations:

- Tasks on critical path are first in order
- Then tasks more distant from the end

Organization of parallel flow

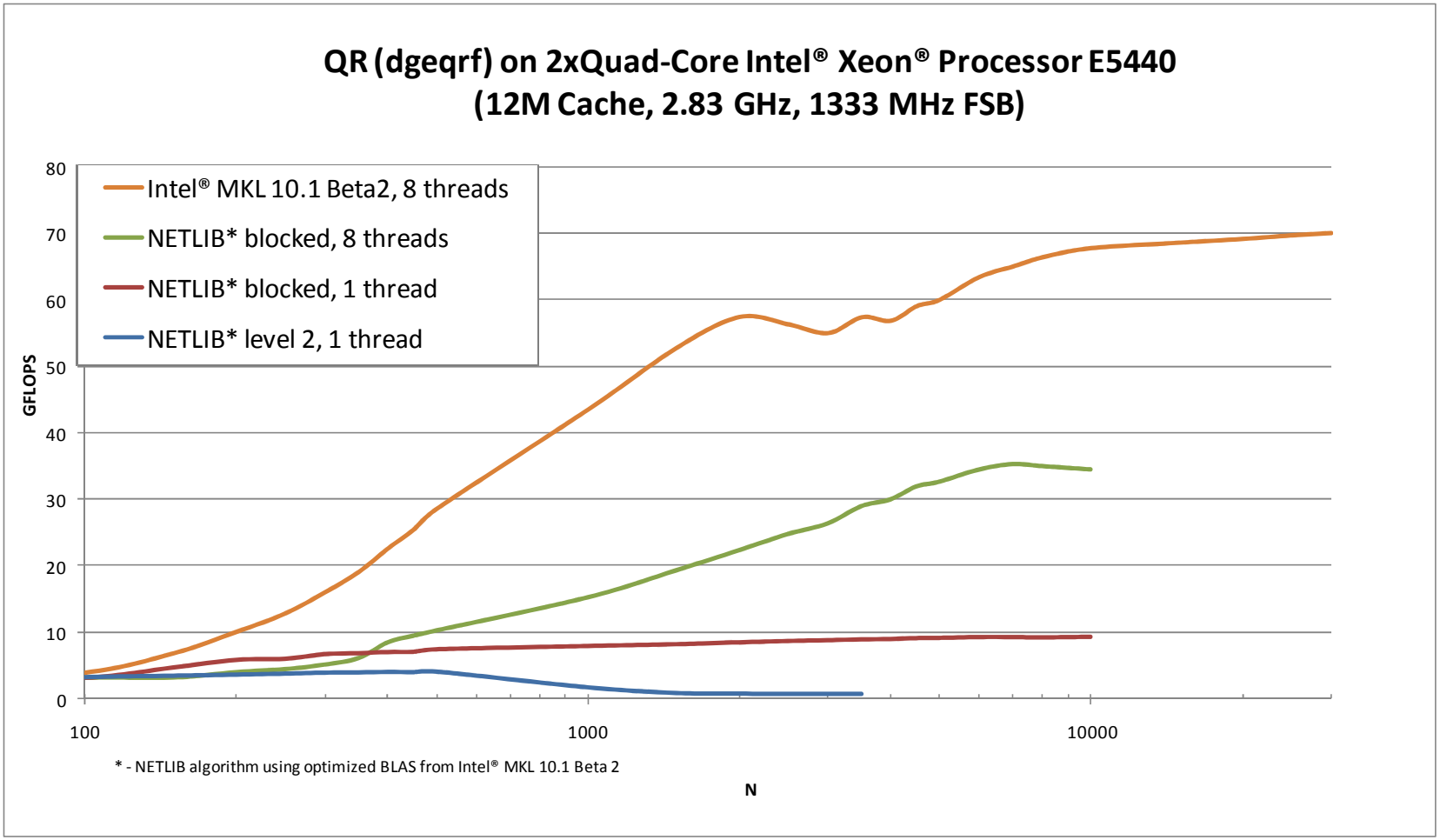
Two roles:

- DAG object – knows what to do.
  - Assigns a task
  - Controls dependencies
  - Optimizes execution flow
- Executive code – knows how to do.
  - Owns parallel constructions
  - Asks an object for tasks
  - Perform assigned task

```
DAG.Init()
parallel do while( DAG.NotFinished() ) private( task )
   critical
        task = DAG.GetTask()
   end critical
   do while( task )
      switch( task ) {
         task1: …
         task2: …
         …:
      }
      task = DAG.CommitTask( task )
    end do
end parallel do
```

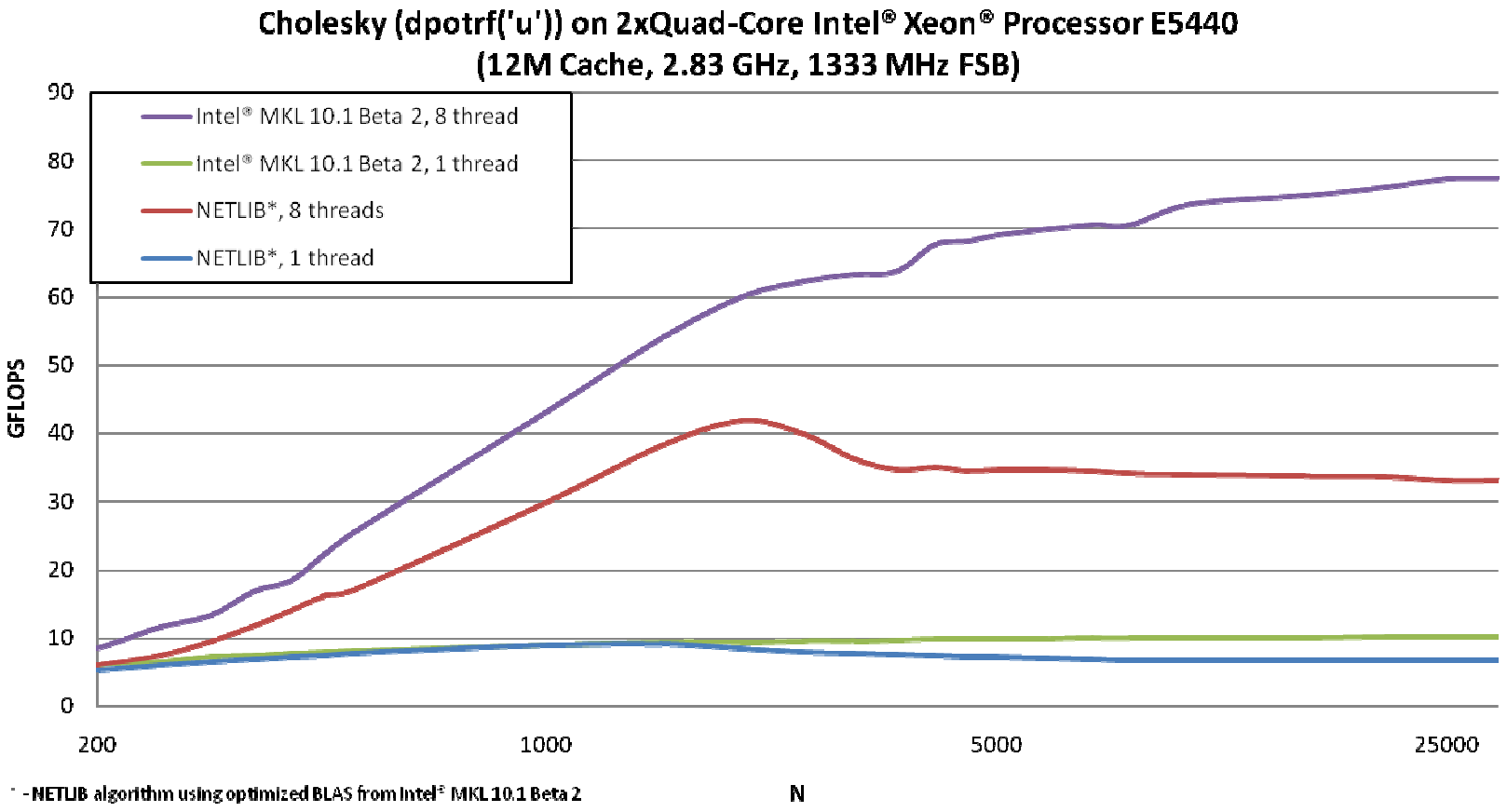
Results

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations



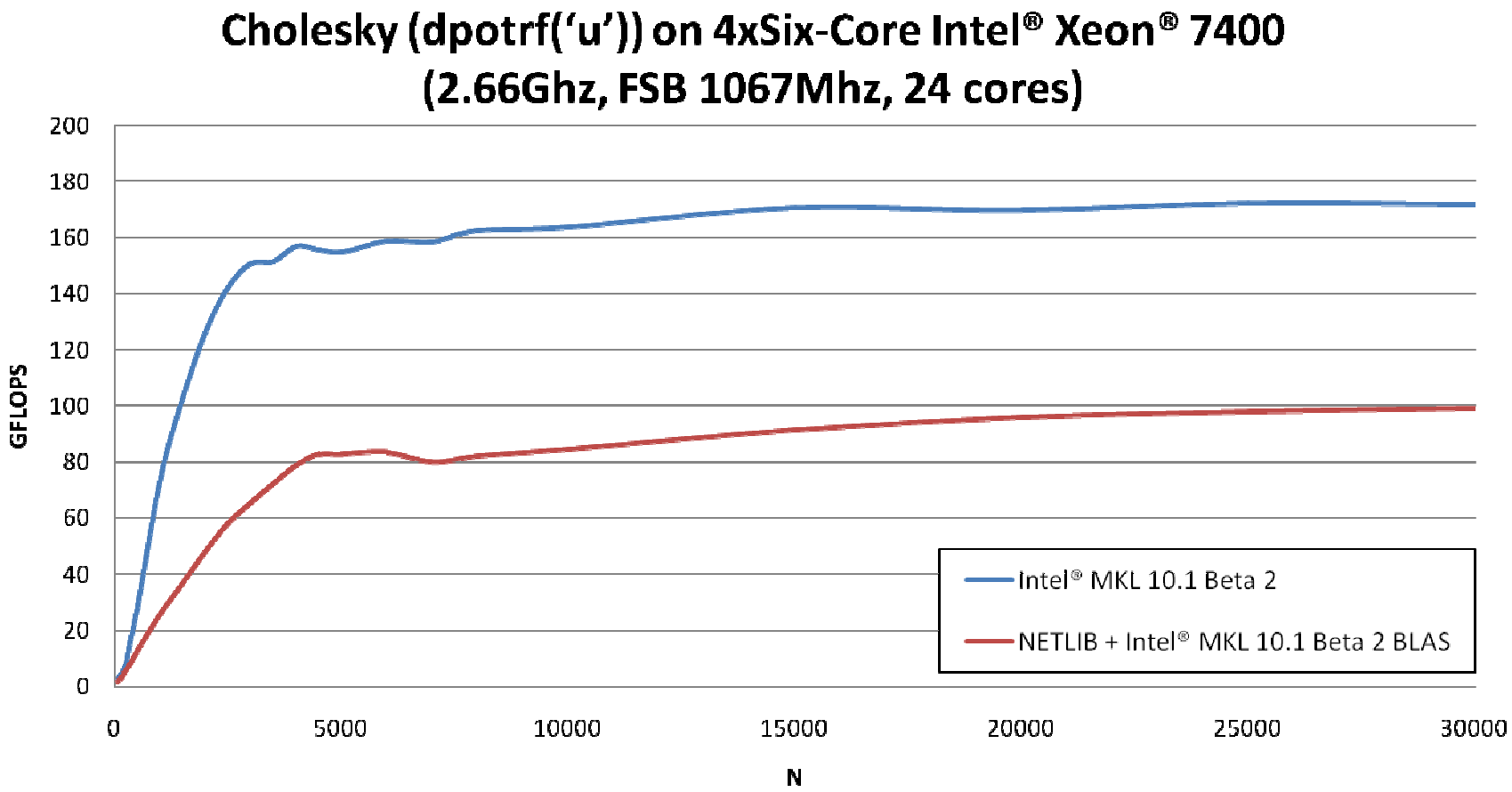
Results

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations



Results

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations



Cholesky scalability

Top speedups (rel. to 1 thread):

4 thr, compact 3.77x

4 thr, scatter **4.12x** (86.3%)

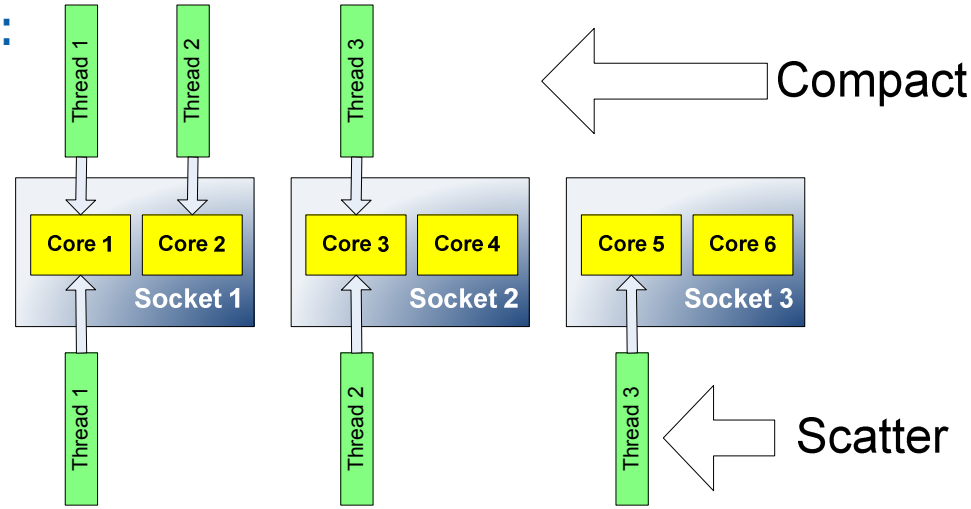
8 thr, compact 6.62x

8 thr, scatter 8.01x

16 thr, compact 14.01x

16 thr, scatter 14.73x

24 thr 20.47x



- Scatter: scale over several sockets – more resources(cache, bandwidth)
- Compact: several cores on one socket – receive less resources
- Full: load gives less bandwidth and cache than have one core => scalability limited if algorithm sensitive to at least one of these

Summary

- Dynamic parallelization implemented well appears to be the best one on real shared memory machines.
- Adoption to dynamical parallelism of Cholesky and QR factorizations allowed to achieve Intel® Math Kernel Library the highest performance results.

Thanks for your attention

Q&A