Blocked Cholesky Factorization with OpenMP* 4.0 Tasks

This is an advanced exercise for creating unstructured parallelism with OpenMP 4.0 tasking. We will use the OpenMP task clause with dependencies to parallelise the processing of a right-looking blocked Cholesky Factorization.

Algorithmic background

Cholesky factorization decomposes an $n \times n$ symmetric positive definite A into the product $A = LL^T$, where L denotes a lower triangular matrix. We assume that the lower triangular part of the matrix A has been stored in a two dimensional array and that at the end of the factorization process it will be overwritten with the matrix L.

To factorize the system in blocks, we first write the matrix A as

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \mathbf{0} \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ \mathbf{0} & L_{22}^T \end{pmatrix},$$

where A_{11} is of the size $n_b \times n_b$, A_{21} is $(n - n_b) \times n_b$ and A_{22} is $(n - n_b) \times (n - n_b)$. Computing the Cholesky factorization of A_{11} block as $A_{11} = L_{11}L_{11}^T$, by equating we get

$$L_{21} \leftarrow A_{21}(L_{11}^T)^{-1}$$
$$\tilde{A}_{22} \leftarrow L_{22}L_{22}^T = A_{22} - L_{21}L_{21}^T$$

and then proceeds recursively for the updated block $\tilde{A}_{22}$. The recursive process, which can be also regarded as block Gaussian elimination, is described in Figure 1.

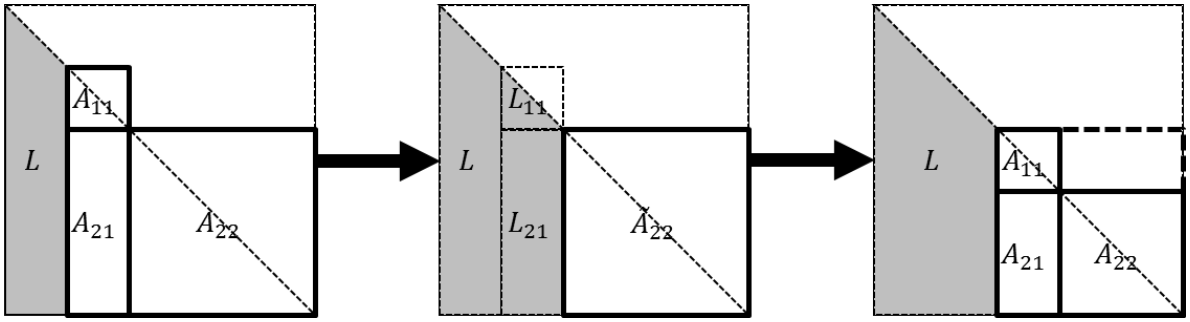

Figure 1 Single step in a computation of a blocked Cholesky Factorization. The computation continues recursively from block $\tilde{A}_{22}$.

In terms of linear algebra operations from BLAS/LAPACK, a single step of the algorithm can be implemented as

1. Compute Cholesky Factorization of A_{11} block with **`DPOTF2`**
2. Compute L_{21} block by solving a linear system with **`DTRSM`**
3. Update the A_{22} block with **`DSYRK`**

Instead of computing L_{21} and A_{22} with a single operation each, the steps 2 and 3 can be split into operations with blocks of the size $n_b \times n_b$ at most. Let there be k subblocks in

total and denote by L_{21}^i the ith subblock of L_{21} and by $A_{22}^{i,j}$ the (i,j)th subblock of the matrix A_{22}. Steps 2 and 3 to compute L_{21} and A_{22} become.

2. For all subblocks i compute $L_{21}^i \leftarrow A_{21}^i (L_{11}^T)^{-1}$ by solving a linear system with **DTRSM**
3. For all subblocks i and $j \leq i$, update the $A_{22}^{i,j} \leftarrow A_{22}^{i,j} - L_{21}^i \, (L_{21}^j)^T$ block with **DGEMM** for $i \neq j$ or **DSYRK** for $i = j$

The concept of subblocking is further illustrated in Figure 2.

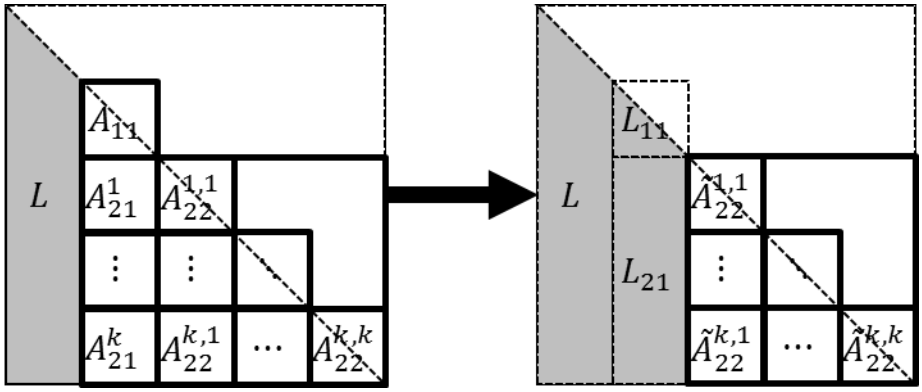

Figure 2 Subblocked computation of L_{21} and A_{22}

The main benefit of using subblocks for computing L_{21} and updating A_{22} is the added parallelism: instead of a single **DTRSM** or **DSYRK** -operation with blocking we have multiple operations the computation of which can be parallelised. A key observation to adding task parallelism is that once A_{11} has been factorized with **DPOTF2**, the computation of $L_{21}^i, i = 1, \cdots, k$ can start. Again, once individual blocks L_{21}^i and L_{21}^j have been computed, the computation of an individual subblock $A_{22}^{i,j}$ can start immediately. This holds also recursively, i.e., once any the diagonal blocks $\tilde{A}_{22}^{i,i}, i = 1, \cdots, k$ have been computed, their factorization with **DPOTF2** can immediately start. Scheduling such dependencies manually is very involved, but becomes straightforward when using a runtime, such as OpenMP 4.0, with built-in tasks and task dependencies.

Building the Application

Decide whether you would like to work with GNU Fortran (**`gfortran`**) or Intel® Fortran compiler (**`ifort`**). The techniques covered in this lab are applicable to both, so you should choose whichever compiler you are most familiar with and have access to.

You can build the **`chol`** application by using the Intel® toolchain with

```
make COMP=intel
```

or with the GNU toolchain with

```
make COMP=gnu
```

Note that the provided **`Makefile`** already includes **`–qopenmp`** or **`–fopenmp`** –flag. However, as there are no OpenMP constructs in the source code the computations runs in serial apart from any internal threading performed by the BLAS/LAPACK library calls.

Note that to compile and link the **`chol`** application, you need to have a working implementation of BLAS and LAPACK available on you system. By default the Intel toolchain will attempt to use Intel® Math Kernel Library for BLAS/LAPACK operations whereas the GNU toolchain will use OpenBLAS. To use some other BLAS or LAPACK, edit the provided **`Makefile`**.

Configuring and Running the Application

For the time being we will concentrate on the **chol** application. Please be patient when doing the first runs. Because of the serial computations, the runs can take some time to complete, especially for large **n**.

Before running the application, be sure to set the OpenMP thread affinity by using the portable OpenMP 4.0 thread affinity syntax as below. The optional **verbose** setting available only with the Intel® OpenMP library will show the binding of threads to logical processors:

```
export OMP_PLACES=threads
export OMP_PROC_BIND=close
[export KMP_AFFINITY=verbose] # Intel OpenMP only
```

Then execute the application **chol**, which takes the matrix size **n** and blocksize **bs** as an input. For instance:

```
./chol 1000 10
```

computes a blocked Cholesky decomposition for a matrix of size **n=1000** with a blocksize **bs=10**. Note that you can make the computation more demanding by increasing **n** and speed up the computation by tuning blocksize **bs**. Run the application now and record the achieved Gigaflops and run time:

Gigaflops/sec:	
Time (s):	

Adding OpenMP Task parallelism

We now add the OpenMP constructs for parallelism, tasking and task dependencies to the **`chol`** application. A serial implementation blocked Cholesky factorization for a given matrix is contained in the source code file **`chol_task.F90`**.

First we spawn a number worker threads by adding an OpenMP **`parallel`** construct that encapsulates the do loop for traversing the all the blocks of rows/columns in the matrix. We then let the **`parallel`** construct to be followed by a **`single`** construct with a **`nowait`** –clause, such that all the compute tasks are added to the runtime by a single thread and that the remaining threads idle at the end of the **`parallel`** section, ready to pick up new tasks.

```
subroutine chol_task(n, A, bs)
  implicit none
  integer, intent(in) :: n
  real(kind=real64) :: A(n,n)
  integer, intent(in) :: bs

  integer :: i, j, k, s, info
  !$omp parallel private(i,j,k,s,info)
  !$omp single
  do k=1,n,bs
  ! …
  end do
  !$omp end single nowait
  !$omp end parallel
```

To parallelize the computation, we now declare each BLAS/LAPACK operation as a single task. To keep the computation correct, either task synchronization through **`taskwait`** construct or declaration of task dependencies are needed. At each step of the algorithm, A_{11} block is factorized, followed by computation of the subblocks of L_{21} and A_{22}.

If explicit synchronization is used, the L_{21} subblocks cannot be computed until the factorization of A_{11} block has been completed. Similarly, before all the subblocks of L_{21} are available, none of the subblocks of A_{22} cannot be updated. In addition, recursive computation from $\tilde{A}_{22}$ block cannot proceed until all its subblocks are ready. Therefore using explicit synchronization causes a fork-join –pattern which forces the threads to wait idle until more computational work is available.

To minimize unnecessary idle time we need to depart from the traditional fork-join model. This is achieved through the use of finer grained tasks and task dependencies. We observe that once the A_{11} block has been factorized, the computation of L_{21} subblocks L_{21}^{i} can begin in parallel for all $i = 1, \cdots, k$. The subblock $A_{22}^{i,j}$ of A_{22} can be updated immediately once the dependant components L_{21}^{i} and L_{21}^{j} are available. Furthermore, computation of the $\tilde{A}_{22}$ block can begin with a factorization of the $\tilde{A}_{22}^{1,1}$ block as soon as it becomes available, followed again by computation of subblocks $\tilde{L}_{21}^{i}$ as soon as their dependencies

are available. Such dependencies can be expressed until the very last block of the computation and form a structure called Directed Acyclic Graph (DAG). DAG can be used by the tasking runtime to maximize the amount of parallelism in the computation and thus minimize the amount of thread idle time.

We now proceed to use OpenMP 4.0 **`task depend`** –clause to express the data dependencies within the blocked Cholesky factorization. Note that in the following text we use a column major Fortran notation for block indices.

First consider the first step of the recursive algorithm, i.e., the factorization of the A_{11} block with **`DPOTF2`**. The factorization can begin as soon as the data for the block is available. Once the factorization is complete, the block of data is overwritten, suggesting an **`inout`** -dependency for the **`(k,k)`** -block.

```
!$omp task depend(inout:A(k,k)) private(s)
  s = min(bs, n-k+1)
  call dpotrf(…)
!$omp end task
```

The second step in the algorithm is the solution of the subblocks L_{21}^{i} with **`DTRSM`**. The computation requires a previously factorized A_{11} block to be available as an input and produces a solved L_{21}^{i} subblock as an output. Thus we have an **`in`** -dependency for the **`(k,k)`** –block and a **`inout`** –dependency for the set of **`(i,k)`** –blocks.

```
!$omp task depend(in:A(k,k)) depend(inout:A(i,k)) private(s)
  s = min(bs, n-i+1)
  call dtrsm(…)
!$omp end task
```

The third and final step of the algorithm is the updating the trailing part of the matrix, i.e., computation of the subblocks $A_{22}^{i,j}$ with **`DGEMM`** for $i \neq j$ and **`DSYRK`** for $i = j$. For $A_{22}^{i,j}$ previously solved subblocks L_{21}^{i} and L_{21}^{j} are needed as an input with an updated $A_{22}^{i,j}$ being produced as an output. For **`(i,k)`** and **`(j,k)`** blocks this yields an **`in`** –dependency and for **`(i,j)`** –block an **`inout`** –dependency.

```
do i=k+bs,n,bs
  do j=k+bs,i-1,bs
  !$omp task depend(in:A(i,k),A(j,k)) depend(inout:A(i,j)) private(s)
    s = min(bs, n-i+1)
    call dgemm(…)
  !$omp end task
  end do
  !$omp task depend(in:A(i,k)) depend(inout:A(i,i)) private(s)
    s = min(bs, n-i+1)
    call dsyrk(…)
  !$omp end task
end do
```


Rebuild the application, run it with the same parameters as previously and record the achieved Gigaflops/sec and the runtime to the table below. Watch out for any "**`*** TEST FAILED ***`**" –messages suggesting that the computed result is different compared to the LAPACK reference. Try running with different values of **`OMP_NUM_THREADS`**, matrix size **`n`** and blocksize **`bs`**. Do you get a good scaling by increasing the number of threads? Do different blocksizes have an impact to the achieved performance?

Gigaflops/sec:	
Time (s):	

Legal Information

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3 and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.

Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Trademark Information

BlueMoon, BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Inside, Cilk, Core Inside, E-GOLD, Flexpipe, i960, Intel, the Intel logo, Intel AppUp, Intel Atom, Intel Atom Inside, Intel CoFluent, Intel Core, Intel Inside, Intel Insider, the Intel Inside logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel Sponsors of Tomorrow., the Intel Sponsors of Tomorrow. logo, Intel StrataFlash, Intel vPro, Intel Xeon Phi, Intel XScale, InTru, the InTru logo, the InTru Inside logo, InTru soundmark, Itanium, Itanium Inside, MCS, MMX, Pentium, Pentium Inside, Puma, skoool, the skoool logo, SMARTi, Sound Mark, Stay With It, The Creators Project, The Journey Inside, Thunderbolt, Ultrabook, vPro Inside, VTune, Xeon, Xeon Inside, X-GOLD, XMM, X-PMU and XPOSYS are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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 go to: http://www.intel.com/design/literature.htm

Software Source Code Disclaimer

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.