Efficient HPL on top of runtime systems

Alycia Lisito

▶ **To cite this version:**

Alycia Lisito. Efficient HPL on top of runtime systems. compas 2024 - Conférence francophone d'informatique en Parallélisme, Architecture et Système, Jul 2024, Nantes, France. hal-04603576

HAL Id: hal-04603576

https://inria.hal.science/hal-04603576

Submitted on 6 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Compas'2024 : Parallélisme / Architecture/ Système
IETR/LS2N - Nantes France, 2-5 juillet 2024

Efficient HPL on top of runtime systems

Alycia Lisito

EVIDEN, Univ. Bordeaux, CNRS, Bordeaux INP, Inria, LaBRI, UMR 5800, F-33400 Talence

---

Résumé
De nos jours, les machines sont de plus en plus grosses, complexes et hétérogènes. Cela rend plus difficile d'avoir un code générique qui soit perfomant, indifféremment de l'architecture de la machine. Afin d'atteindre cet objectif, de plus en plus d'algorithmes sont implémentés sur des supports d'exécution à base de tâches. Dans ce papier, nous proposons une implémentation de HPL sur ces supports d'execution à base de tâches. Pour cela, nous avons implémenté l'algorithme dans CHAMELEON, qui est une bibliothèque déjà optimisée et performante, avec le support d'exécution STARPU. Nous expliquons comment arriver à une implementation performante et montrons comment nous arrivons à obtenir 93% en pic de performance par rapport au code de référence sans pivotage.

---

1. Introduction

Nowadays, modern machines embed several computing hardware solutions. Indeed, 9 of the 10 fastest supercomputers in the world use nodes holding both CPUs and GPUs [6]. However, this heterogeneity implies a higher implementation cost. Exploiting efficiently all the available computing resources requires to implement several levels of parallelism. Moreover, each material comes with a large number of software solutions. This raises the issue of portability and maintainability of hybrid parallel codes. Over the past two decades, the task paradigm has frequently been shown as a solution to address portably and efficiently modern computing architectures. The emergence of several libraries, implementing task parallelism, has indeed often brought improvements, particularly concerning applications based on linear algebra [1, 3, 5, 11, 13].
Supercomputers are ranked worldwide into a list called TOP500 [6]. The ranks are evaluated with a benchmark called High Performance Linpack (HPL [7]). Algebraically, this benchmark aims at solving the equation $Ax = b$, where x is the unknown, A is an invertible matrix problem and b is a given right hand side. To this end, the benchmark implements a direct LU decomposition method with partial pivoting. There are several highly optimized implementations of the HPL benchmark but, to our knowledge, none of these are implemented on task based runtime systems to provide performance portability on various architecture hardware.
Our goal is to propose an implementation of the HPL benchmark on top of a runtime system based on the task paradigm. In this paper, we focus on one of the principal challenges, which resides in designing an efficient task based version of the partial pivoting algorithm, as it raises both problems of granularity and low arithmetic intensity. As a main contribution, we propose to increase the computational intensity and to modify the granularity of panel operations by coupling an inner-blocking strategy to a task batching mechanism.
The HPL algorithm is presented and the granularity challenge is motivated in Section 2. Then,

proposed modifications are described and evaluated in Section 3. Finally, the state of the art is given by Section 4.

2. HPL algorithms and challenges

In order to solve the equation $Ax = b$, the matrix A is factorized into the product of two lower and upper triangular matrices L and U (GEneral TRiangular Factorization). There are different ways to factorize A, we will first talk about the blocked factorization without pivoting to simplify the algorithm. The GETRF algorithm consists of three nested loops as shown by Algorithm 1. At each iteration k (k going through the blocks and nt being the number of blocks per row of the matrix), scalar operations are applied to the blocks. First a factorization GETRF is performed on the diagonal block $A(k,k)$, then the blocks in the column and row k are solved with the TRSM operation, and finally, the rest of the matrix is updated with GEMMs.
In the scalar GETRF, every diagonal element is used to divide their columns, which can lead to numerical errors if one of these element is close to zero. In order to obtain a numerical stability, each element of the diagonal is permuted with the greatest element of the column called the pivot. This is called partial pivoting.

```
Data: k, A(k:nt, k)
Result: A(k:nt, k) ← L(k:nt, k)U(k, k)
for k = 0 to nt − 1 do
    A_{k,k} ← GETRF(A_{k,k})
    for n = k + 1 to nt − 1 do
        A_{n,k} ← TRSM_RU(A_{k,k}, A_{n,k})
        A_{k,n} ← TRSM_LL(A_{k,k}, A_{k,n})
    end
    for n = k + 1 to nt − 1 do
        for m = k + 1 to nt − 1 do
            A_{m,n} ←
              GEMM(A_{m,k}, A_{k,n}, A_{m,n})
        end
    end
end
```

Algorithm 1: GETRF factorization without partial pivoting algorithm.

```
Data: k, A(k:nt, k)
Result: A(k:nt, k) ← P_k L(k:nt, k)U(k, k)
P_k ← Id
for c = 0 to nb − 1 do
    for m = k to mb − 1 do
        [ipiv(k), mpiv] ←(reduce)— SEARCH_MAX(A_{m,k}, c)
    end
    [A_{k,k}, A_{mpiv,k}] ← SWAP_ROWS(A_{k,k}, A_{mpiv,k}, c, ipiv(k))
    for m = k to mb − 1 do
        A_{m,k} ← SCALE_AND_UPDATE(A_{m,k}, A_{k,k}, c)
    end
end
P_k ← IPIV_TO_PERM(ipiv(:))
```

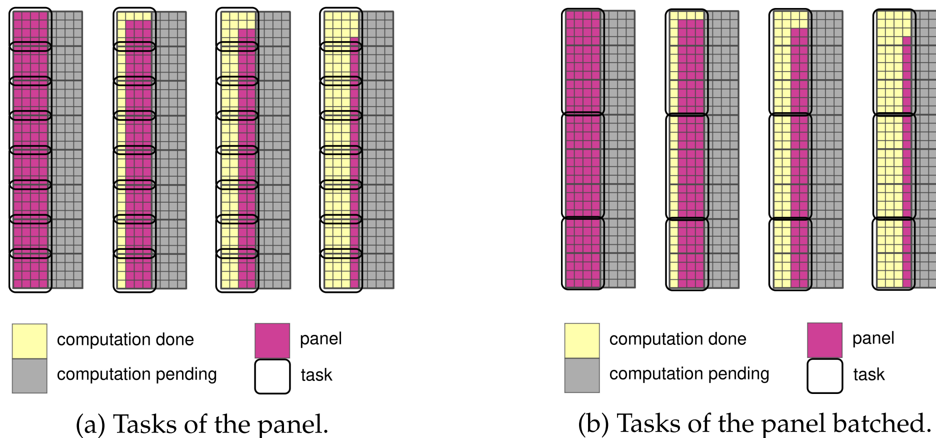
Algorithm 2: Factorization of panel k for the block LU algorithm with partial pivoting algorithm.

Partial pivoting forces us to go through every column of every block in order to find every pivot. The GETRF of $A(k,k)$ and the TRSM_RU of $A(k,n)$ (see Algorithm 1) are therefore, replaced by the loop c in Algorithm 2, this is the panel operation. The pivot is looked for, in each column of every column of the panel, which creates a reduction (REDUX) operation between the workers. The REDUX operation consists in finding the maximum of the column (the pivot) with a reduction between the rows. Once the pivot is found, the diagonal row is swapped with the pivot's row. Then, the blocks of the column k are updated with outer-product (BLAS 2 operations).
The algorithm previously *block-wise* becomes then *block-wise* $\times$ *column-wise*, which generates two main issues : the size of the tasks and the number of tasks generated. Indeed, the BLAS 2 operations have a smaller granularity than the BLAS 3 operations (TRSM_RU without the partial pivoting) which creates very small tasks, and therefore, reduces the number of flops per task. As for the number of tasks generated, instead of mt tasks per panel, $mt \times nb$ tasks are created per panel (mt being the number blocks of per column of the matrix and nb the number

of columns in a block). This significantly inscreases the risk of performance loss because of the runtime overhead.
Moreover, the REDUX operation per column creates synchronization barriers. These synchronization barriers slow down the algorithm at each column iteration as every worker has to wait for each other and the REDUX is done between two workers at once. This makes the panel hard to handle with a runtime.
The STARPU runtime recommends that the execution time of a task should be between 1ms and 10ms on CPUs in order to maximize its efficiency [12]. The average execution time of a panel task is 0.1ms which is too small for STARPU and for the target task size of HPL.

3. Inner blocking and batching the tasks on top of STARPU

The performance of the first implementation done of the HPL with STARPU on CHAMELEON was far from the targeted performance (the performance of the GETRF without pivoting). This implementation, called *ppivpercolumn*, did not have any optimization. The performance peak was at 1.25TFlop/s with the targeted performance at 1.65TFlops/s as shown by Figure 2.
A first solution to reduce the impact of the panel is to use inner blocking. Inner blocking introduces a new level of block division : the panel is split into tasks of size $\text{nb} \times \text{ib}$ (with ib the size of the inner blocks). This allows us to limit the use of level 2 BLAS operations on each sub-block of the panel and use level 3 BLAS operations to update the rest of the panel. The level 2 BLAS operations are done on blocks of size $\text{nb} \times \text{ib}$ instead of $\text{nb} \times \text{nb}$ and the level 3 BLAS are done on the remaining part of the blocks every ib iteration. Hence, the granularity of the panel tasks increases which generates more flops and brings us closer to the HPL target.
Another solution, on top of the inner blocking, is to group some blocks of the panel together in order to generate fewer tasks. To implement this, instead of submitting one task per blocks during the panel, we set a batch size b and submit one task per b blocks. This reduces the number of tasks generated by b and they are b times bigger which reduces the runtime overhead.



(a) Tasks of the panel. (b) Tasks of the panel batched.

FIGURE 1 – Example of a panel operation on a matrix 32×32 with blocks of size 4×4.

The example in Figure 1 shows the first panel done on a matrix 32×32 with blocks of size 4×4. The algorithm without batching generates 32 tasks (black squares Figure 1a) with an average of 21 flops per task and the algorithm with a batching of 3 generates 12 tasks (black rectangles

Figure 1b) with an average of 55 flops per task. We can see with this small example that the number of tasks is almost reduced by 3 and the average flops per task is almost increased by 3.
The experiments were conducted on one `diablo` cluster of the Plafrim [1] platform. One `diablo` node is equipped with two AMD Zen3 EPYC 64-cores running at 2.45 GHz and 1 TB of memory. The environment was handled with the modules available on Plafrim : Intel MKL 2020 for BLAS kernels, gcc 12.2 and StarPU 1.4.4 was compiled.
The experiments are based on the tag *compas2024_batch_getrf* of Chameleon available on the public git repository [2].

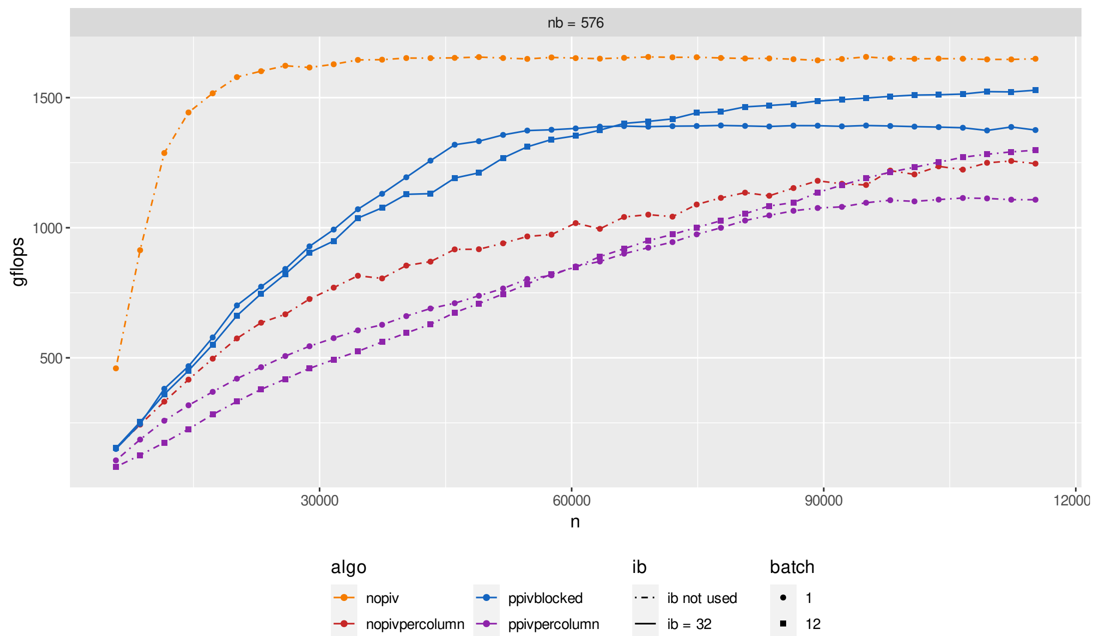


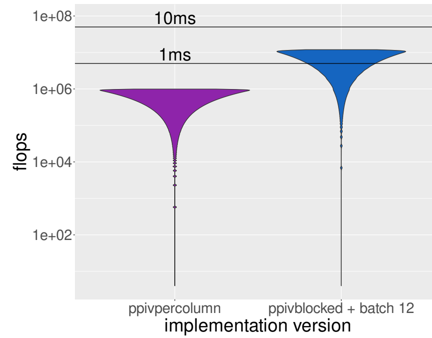
Figure 2 – Comparaison of the performances of the different algorithms

Figure 2 compares the performance of the optimised HPL without pivoting (*nopiv* in orange), the naive implementation of HPL with (*ppivpercolumn* in purple) and without (*nopivpercolumn* in red) the pivoting and the optimised implementation of HPL with the inner blocking (*ppviblocked* in blue). The batching is represented with the square dots. Size 12 was chosen through experiments on the `diablo` cluster.
The *nopivpercolumn* is significantly less effective than the *nopiv*. This proves that going through every column of every block and using BLAS 2 operations instead of BLAS 3 is far less efficient even without the redux operations. The *ppivpercolumn* is less efficient than the *nopivpercolumn*, but the batching manages to be more effective on large matrices. As expected, the *ppviblocked* is better than the *ppivpercolumn* thanks to the inner blocking. The batching of the *ppviblocked* is more effective with matrices larger than 70000 but shows no improvement on smaller matrices. There is room for improvement on smaller matrices, with a batch size chosen dynamically for instance. The peak of performance of the *ppviblocked* with the batch is 1.53TFlop/s which is close to the peak of the *nopiv*, however the peak is reached slowly.

1. https ://www.plafrim.fr
2. https ://gitlab.inria.fr/solverstack/chameleon

	ppivpercolumn	*ppviblocked batch 12*
Tasks nbr	11 482 299	1 020 100
Time (ms/task)	0.151703	1.079328
Flops/task	494 863	5 625 992

(a) Panel tasks statistics of a matrix of size 115200 with a block size of 576



(b) Tasks size repartition

FIGURE 3 – Panel tasks statistics of a matrix of size 115200 with a block size of 576, with the naive (*ppivpercolumn*) and the optimised (*ppviblocked* + *batch 12*) algorithms

As shown in Table 3a, the batching size 12 reduces the number of tasks by 11.25 and increases the average task size by 11.37. Thanks to the batching, the average execution time is $1.08ms$ instead of $0.15ms$ which fits into the STARPU recommendations. Moreover, the task size repartion represented in Figure 3b indicates that the majority of the tasks are between $1ms$ and $10ms$.

4. Related work

There are lots of existing HPLs, the netlib HPL which is open source and several private HPLs : `intel` HPL, AMD CPU HPL and GPU rocHPL [4] and `nvidia` HPL. They all have optimizations such as the inner blocking and the lookahead but none of them use runtimes which makes these HPLs not easily portable. There are other pivoting algorithms such as the tournament pivoting implemented in `Slate` [2] but this algorithm is not numericaly stable and therefore not considered as HPL. Our HPL is numericaly stable and easily portable as we use runtimes.
Several instances of batching exist, the BLAS library uses batching within the kernel [8]. There has also been a batching of tasks tested in STARPU [10]. However, those batchings are not at application level like the batching we implemented. Our batching gives us more control over the tasks we batch.
More and more linear algebra libraries use runtimes [1, 3, 5, 11, 13] but only a few of them implement HPL. HPL is implemented in *dplasma* with *parsec*, this HPL uses inner blocking but no batching. The HPL implemented in CHAMELEON is the only one which uses runtimes, inner blocking and batching.

5. Conclusion

In this paper, we have shown that we can implement an efficient HPL on top of a runtime system thanks to batching. However, a static batch size is not the best solution as it does not yield the best results on small matrices. For our future work, we will implement a batching with a size chosen dynamically according to the state of the factorization and the number of workers available. Another idea would be to exploit the STARPU recursive tasks [9], in order to manage the task granularity automatically. Indeed, the STARPU recursive tasks allows us to take the decision to refine, or not, the granularity dynamically. Finally, an improvement on the REDUX operation can be made. We will look into a more efficient way to communicate

the pivot in order to avoid doing a reduce and then a broadcast, in particular do an allreduce instead.

Aknowledgement

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https ://www.plafrim.fr).

Bibliographie

1. Agullo (E.), Augonnet (C.), Dongarra (J. J.), Ltaief (H.), Namyst (R.), Thibault (S.) et Tomov (S.). – A hybridization methodology for high-performance linear algebra software for gpus. *In : GPU Computing Gems Jade Edition*, pp. 473–484. – Elsevier, 2012.
2. Alomairy (R.), Gates (M.), Cayrols (S.), Sukkari (D.), Akbudak (K.), YarKhan (A.), Bagwell (P.) et Dongarra (J.). – *Communication Avoiding LU with Tournament Pivoting in SLATE*. – Rapport technique n18, ICL-UT-22-01, 2022-01 2022.
3. Buttari (A.), Langou (J.), Kurzak (J.) et Dongarra (J. J.). – A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, vol. 35, n1, janvier 2009, pp. 38–53.
4. Chalmers (N.), Kurzak (J.), McDougall (D.) et Bauman (P. T.). – Optimizing high-performance linpack for exascale accelerated architectures, 2023.
5. Chan (E.), Van Zee (F. G.), Bientinesi (P.), Quintana-Orti (E. S.), Quintana-Orti (G.) et Van de Geijn (R.). – Supermatrix : a multithreaded runtime scheduling system for algorithms-by-blocks. – In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 123–132. ACM, 2008.
6. Dongarra (J.). – Performance of various computers using standard linear equations software. *Computer Technical Report Number CS - 89 – 85*, vol. 20, 11 2002.
7. Dongarra (J. J.). – Performance of various computers using standard linear equations software. *ACM SIGARCH Computer Architecture News*, vol. 20, n3, 1992, pp. 22–44.
8. Dongarra (J. J.), Duff (I. S.), Gates (M.), Haidar (A.), Hammarling (S.), Higham (N. J.), Hogg (J.), Lara (P. V.), Luszczek (P.), Zounon (M.), Relton (S. D.), Tomov (S.), Costa (T. B.) et Knepper (S.). – Batched blas (basic linear algebra subprograms) 2018 specification. – 2018.
9. Faverge (M.), Furmento (N.), Guermouche (A.), Lucas (G.), Namyst (R.), Thibault (S.) et Wacrenier (P.-a.). – Programming Heterogeneous Architectures Using Hierarchical Tasks. *Concurrency and Computation : Practice and Experience*, vol. 35, n25, 2023.
10. Rossignon (C.), Pascal (H.), Aumage (O.) et Thibault (S.). – A NUMA-aware fine grain parallelization framework for multi-core architecture. – In *PDSEC - 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing - 2013*, Boston, United States, mai 2013.
11. Song (F.), YarKhan (A.) et Dongarra (J. J.). – Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. – In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, *SC '09*, SC '09, pp. 19 :1–19 :11, New York, NY, USA, 2009. ACM.
12. Thibault (S.). – *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. – Habilitation à diriger des recherches, Université de Bordeaux, décembre 2018.
13. Tomov (S.), Dongarra (J. J.) et Baboulin (M.). – Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, vol. 36, n5-6, juin 2010, pp. 232–240.