Journal of Parallel and Distributed Computing 175 (2023) 51–65

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Programming parallel dense matrix factorizations and inversion for new-generation NUMA architectures

Sandra Catalán [a], Francisco D. Igual [a], José R. Herrero [b], Rafael Rodríguez-Sánchez [a], Enrique S. Quintana-Ortí [c,*]

[a] Departamento de Arquitectura de Computadores y Automatica, Universidad Complutense de Madrid, Madrid, Spain
[b] Departament d'Arquitectura de Computadors, Universitat Politecnica de Catalunya, Barcelona, Spain
[c] Departamento de Informatica de Sistemas y Computadores, Universitat Politecnica de Valencia, Valencia, Spain

ARTICLE INFO

Article history:
Received 14 April 2022
Received in revised form 29 December 2022
Accepted 5 January 2023
Available online 13 January 2023

Keywords:
NUMA architectures
Chiplets
Dense linear algebra
Shared memory programming
Portability

ABSTRACT

We propose a methodology to address the programmability issues derived from the emergence of new-generation shared-memory NUMA architectures. For this purpose, we employ dense matrix factorizations and matrix inversion (DMFI) as a use case, and we target two modern architectures (AMD Rome and Huawei Kunpeng 920) that exhibit configurable NUMA topologies. Our methodology pursues performance portability across different NUMA configurations by proposing multi-domain implementations for DMFI plus a hybrid task- and loop-level parallelization that configures multi-threaded executions to fix core-to-data binding, exploiting locality at the expense of minor code modifications. In addition, we introduce a generalization of the multi-domain implementations for DMFI that offers support for virtually any NUMA topology in present and future architectures.

Our experimentation on the two target architectures for three representative dense linear algebra operations validates the proposal, reveals insights on the necessity of adapting both the codes and their execution to improve data access locality, and reports performance across architectures and inter- and intra-socket NUMA configurations competitive with state-of-the-art message-passing implementations, maintaining the ease of development usually associated with shared-memory programming.

1. Introduction

Power, performance, area, cost and time-to-market have been key drivers for the adoption of heterogeneous integration technologies. As Moore's Law scaling is nearing its end, the traditional monolithic silicon chip, for which a flaw in one part can make the entire device unusable, is being abandoned in favor of systems-on-chip (SoC), composed of multiple small *chiplets* leading to less complex integrated circuits, which can be built in the most efficient manufacturing process according to their characteristics. Advanced packaging technologies and substrate design, which allows for much higher bandwidth between chiplets, have enabled integrating chiplets from different manufacturing process flows into a single package [2,11,22,26]. Thus, disintegrating complex chips improves yield and reduces costs, while accommodating more easily specialized systems, with the net result of being nowadays adopted by most major hardware manufacturers in their current microprocessor designs [29,30,48]. In addition, industry leaders in semiconductors, packaging, IP suppliers, foundries, and cloud service companies are currently standardizing an open chiplet ecosystem: the Universal Chiplet Interconnect Express (UCIe) for integrating chiplets in future semiconductor designs [44]. This hardware disaggregation paradigm matches the high configurability needed in some scenarios. For instance, when targeting multi-tenant scenarios, (either multi-application, multi-container or multi-VM –Virtual Machine–), resource management capabilities provide a fine-grain mechanism to monitor, handle and reduce resource contention, improving the efficiency and predictability of executions [25,31,50]. However, portability on such novel architectures becomes a challenge, due to their complex and configurable memory hierarchies.

These new programmability and performance portability challenges directly translate into complex and fine-grained application-level adaptation, with significant impact on underlying scientific libraries, in which many applications and frameworks delegate to obtain performance. Among them, dense linear algebra libraries in general, and matrix factorization routines in particular, are the foundation in the quest for performance and scalability in novel applications in science and engineering; in the afore-described sce-

* Corresponding author.
E-mail address: quintana@disca.upv.es (E.S. Quintana-Ortí).

https://doi.org/10.1016/j.jpdc.2023.01.004

nario, these fundamental libraries will suffer from the rapid evolution on the complexity of architectures, and hence techniques and methodologies for rapid adaptation will become mandatory.

1.1. Configurable NUMA memories

Memory is a key shared resource across applications, with a critical impact on performance. In the last years, the *memory wall* [28,47] has been tackled via NUMA (Non-Uniform Memory Access) architectures together with multi-socket platforms [23,35]. Unfortunately, this comes at the cost of increasing the design space and introducing a considerable burden on the programmers' shoulders, who now have to avoid remote memory accesses as well as to control thread-to-core pinning [21,33,38]. To partially alleviate this situation, NUMA-aware optimizations have been introduced in most levels of the software stack, including applications [13,43,49], libraries and middleware [32,36], hardware-software co-design of runtime and operating systems [9,24,39], hypervisors [46], and container orchestrators [16].

The NUMA configuration of recent architecture designs from AMD (e.g., Zen2) [30] and Huawei (Kunpeng) [48] can be modified off-line at boot time. An appropriate selection of the NUMA scheme, depending on the server target applications, is therefore crucial to achieve the goals of isolation and contention control. Unfortunately, this hampers performance portability for parallel codes that need to span across multiple NUMA nodes, on the same or across different sockets.

As a motivational example, consider the results in Fig. 1. The experiment was carried out on Rome, a multi-core AMD-based server equipped with two sockets and 64 cores per socket.[1] The experiment computes an LU factorization with partial pivoting [18] for a large square matrix dimension (of order 30720) using a LAPACK-style coding [5] and parallelization scheme, where all multi-threaded parallelism is extracted from within the Basic Linear Algebra Subprograms (BLAS) [15]. The number of BLAS threads varies between 1 and 128. The matrix is generated and remains on the first NUMA node during the complete execution, and we configure the server with three distinct numbers of NUMA nodes per socket (NPS): 1 (red line), 2 (blue line), and 4 (green line). This experiment illustrates that the scalability heavily degrades as the number of threads span across the two physical sockets, as would be expected on a typical NUMA multi-socket setup. In addition, the possibility of configuring different number of NUMA NPS implies that the performance variability depends on the specific setup, and yields a variety of scalability degrees depending on the selected NUMA topology. Ideally, multi-threaded applications should be flexible enough to handle this variation in the hardware topology with minimal modifications to their source code.

1.2. Contributions

In this paper, we address the programmability and performance portability challenges introduced by reconfigurable NUMA architectures in new generation processors for a specific domain, dense linear algebra (LA), demonstrating that it is possible to simultaneously integrate NUMA-awareness into popular algorithms for dense matrix factorizations and inversion (DMFI) while still controlling the inherent complexity of code development for this type of architectures. In more detail, we make the following contributions:

- We expose the significant performance penalty introduced by NUMA-oblivious implementations on current servers.

[1] The complete details of the Rome platform are provided in Section 4.1.

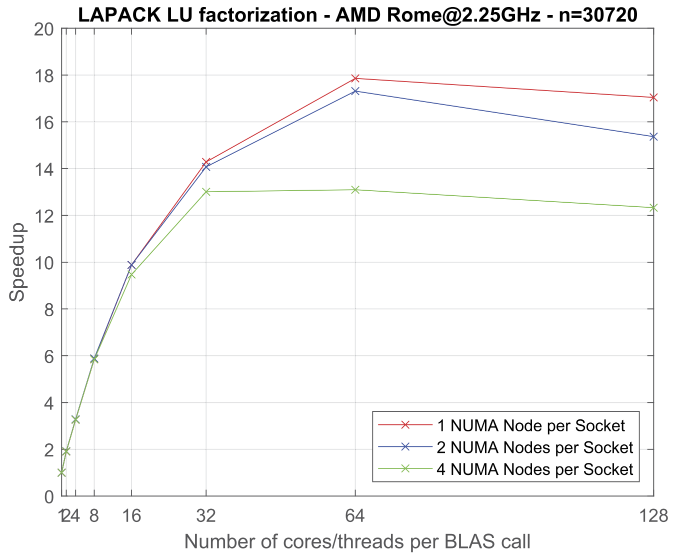


Fig. 1. Speed-up for an LU factorization with partial pivoting on a dual-socket AMD Zen2 architecture (Rome) for different configurations of NUMA nodes per socket (NPS). The speed-up is calculated with respect to a sequential implementation running on a single core of the machine.

- We demonstrate that a high-level approach towards designing LA algorithms can substantially alleviate the development effort for the programmer, while increasing performance in situations in which LA algorithms span across multiple NUMA domains. Concretely, we provide a generic NUMA-aware routine for DFMI that comprises only 32 lines of code (including comments).
- We validate our approach via a complete experimental evaluation of three DMFI kernels on two state-of-the-art dual-socket NUMA architectures, namely: an AMD Rome server with 128 cores and up to 16 NUMA NPS, and an ARM-based Huawei Kunpeng server with 96 cores and 2 NUMA NPS.

The rest of the paper is structured as follows. Section 2 provides a common algorithmic framework for DMFI that will be the baseline for NUMA-aware implementations. Section 3 provides the necessary modifications on the common algorithmic framework to adapt it to NUMA architectures. Section 4 evaluates the NUMA-aware DMFI implementations on two different NUMA architectures, and places the shared-memory implementations into context, comparing them with state-of-the-art distributed-memory codes. Section 5 closes the paper with a number of relevant concluding remarks and avenues for future work.

2. Parallel dense matrix factorizations and inversion

In this section, we offer a brief review of blocked LA algorithms for DMFI, and describe three main options to exploit thread-level parallelism for these types of operations on multicore architectures.

For the presentation of all our LA algorithms, in the remainder of the paper we will consider an $m \times n$ matrix A where, for simplicity, we assume that m, n are both integer multiples of the algorithmic block size b. Furthermore, we will consider a partition of A into $m_b \times n_b = m/b \times n/b$ blocks, each of dimension $b \times b$. In our Matlab-like notation, $A(c_1 : c_2, d_1 : d_2)$ denotes the submatrix of A that spans the intersection between the row-blocks $c_1, c_1+1, \ldots, c_2$ and the column-blocks $d_1, d_1+1, \ldots, d_2$, comprising the entries in the intersection of rows $(c_1-1)\cdot b+1, (c_1-1)\cdot b+2, \ldots, c_2\cdot b$ and columns $(d_1-1)\cdot b+1, (d_1-1)\cdot b+2, \ldots, d_2\cdot b$ of the matrix. Also, matrix indices start at 1. For simplicity, we

```
void DMFI( matrix A, int n_b )
{
  for ( int k = 1; k <= n_b; k++ ) {
    // Factorize panel k
    PF( A(:,k) );
    // Update panels 1:k−1 w.r.t. panel k
    LU( A(:,k), A(:,1:k−1) );
    // Update panels k+1:n_b w.r.t. panel k
    TU( A(:,k), A(:,k+1:n_b) );
  }
}
```

Listing 1: Simplified routine for DMFI.

define the final row/column block of a matrix with the keyword *"end"*.

2.1. Common algorithmic skeleton

Listing 1 displays a blocked algorithm for a "generic" matrix factorization expressed with a high level of abstraction. At the k-th iteration of the loop, the algorithm first computes the factorization of the k-th column-block of the matrix via routine PF (for panel factorization), to then update the leading and trailing submatrices (that is, the blocks to the left and right of the k-th column-block) respectively via routines LU and TU (for leading and trailing updates, respectively).

The algorithmic skeleton in Listing 1 accommodates a number of matrix operations for the solution of linear systems, including the LU and QR factorizations, as well as matrix inversion via Gauss-Jordan elimination [18]. For example, for the LU factorization, PF decomposes (the diagonal and subdiagonal blocks in) the "current" (i.e., the k-th) panel into the product of a unit lower triangular matrix L and an upper triangular matrix U, integrating partial pivoting for numerical stability [18]:

$$P_k \begin{bmatrix} A(k,k) \\ A(k+1:m_b,k) \end{bmatrix} = \begin{bmatrix} L(k,k) \\ L(k+1:m_b,k) \end{bmatrix} U(k,k); \tag{1}$$

LU then simply applies the row permutations dictated by the pivoting scheme, in P_k, to the leading submatrix:

$$\begin{bmatrix} A(k,1:k-1) \\ A(k+1:m_b,1:k-1) \end{bmatrix}; \tag{2}$$

and TU applies the same row permutations to the trailing submatrix (omitted for brevity), followed by the (unit) lower triangular solve:

$$U(k,k+1:n_b) \;:=\; L(k,k)^{-1} \cdot A(k,k+1:n_b) \tag{3}$$

and the submatrix update:

$$\begin{aligned} A(k+1:m_b,k+1:n_b) &:= A(k+1:m_b,k+1:n_b) \\ &\quad - L(k+1:m_b,k) \cdot U(k,k+1:n_b). \end{aligned} \tag{4}$$

In rough detail, for the QR factorization, PF decomposes the diagonal and subdiagonal blocks of the current panel into the product of an orthogonal matrix Q and an upper triangular factor R (though the orthogonal matrix is rarely built explicitly); TU applies the orthogonal transforms to the trailing submatrix, and LU does not perform any operation. For matrix inversion, PF decomposes the current panel via Gauss-Jordan transforms, while LU and TU apply the corresponding row permutations and transforms to the leading and trailing submatrices.

In general, the realizations of these three matrix operations overwrite their input matrix operand with their corresponding outputs. Also, choosing a "sufficiently" large value b for the blocked procedure in Listing 1 raises the ratio between floating-point operations and memory accesses, in principle improving performance.

```
void DMFI_TP( matrix A, int n_b )
{
  #pragma omp parallel
  #pragma omp single
  for ( int k = 1; k <= n_b; k++ ) {
    // Factorize panel k
    #pragma omp task depend( inout: A(:,k) )
    PF( A(:,k) );

    // Update panels 1:k−1 w.r.t. panel k
    for ( int j = 1; j < k; j++ )
      #pragma omp task depend( in: A(:,k),
                               inout: A(:,j) )
      LU( A(:,k), A(:,j) );

    // Update panels k+1:n_b w.r.t. panel k
    for ( int j = k+1; j <= n_b; j++ )
      #pragma omp task depend( in: A(:,k),
                               inout: A(:,j) )
      TU( A(:,k), A(:,j) );
  }
}
```

Listing 2: Simplified routine for DMFI, with TP extracted via OpenMP tasks.

Hereafter we will abstract ourselves from these internal details and the practical implementation of the operations in (1)–(4), as they are not relevant for the techniques described in this work.

2.2. Multi-threaded parallelizations

MULTI-THREADED BLAS. The conventional (or default) parallel realization of a DMFI in LAPACK [5] extracts all parallelism from a multi-threaded instance of the BLAS [15]. In turn, the kernels of such a multi-threaded BLAS internally exploit loop parallelism, and they are implemented as blocked algorithms that carefully re-use some parts (blocks) of the matrix to take advantage of the processor cache hierarchy [19,27,45]. For architectures with a small number of cores, this approach provides a portable solution with fair performance.

TASK PARALLELIZATION VIA A RUNTIME. In the last years, a pure task-parallel (TP) solution for LA algorithms has been investigated in a number of projects [1,6,14,34]. In these efforts:

1. TP is explicitly exposed by dividing the LU and TU operations into multiple finer-grain blocks/tasks;
2. Task dependencies are annotated with the appropriate directives/clauses; and
3. The result is passed to a runtime system that orchestrates the parallel execution.

Listing 2 exemplifies the application of this approach to the generic algorithm for DMFI, with tasks annotated using the OpenMP task construct.
HYBRID PARALLELIZATION. A third approach, which combines task-parallelism with BLAS-level loop-parallelism [10,12], renders an alternative with significant performance advantages over the two previously described in this section. For our generic skeleton for DMFI, this hybrid approach can be formulated, using OpenMP, as illustrated in Listing 3. There, the OpenMP sections construct is equivalent to a task parallelization when there are no inter-task dependencies. In addition, the LU and TU "tasks" internally invoke a multi-threaded instance of BLAS, yielding the hybrid (or two-level) parallelization scheme.

Our solution does not employ the concept of task in the OpenMP "sense". Instead, our tasks refer to the "individual routines" which are encountered during the execution of the algorithm, such as (PF, LU, TU, etc.). Furthermore, our algorithms do not rely

```
void DMFI_HP( matrix A, int n_b )
{
  for ( int k = 1; k <= n_b; k++ ) {
    // Factorize panel k
    PF( A(:,k) );
    #pragma omp parallel sections num_threads(2)
    {
      #pragma omp section
      // Update panels 1:k-1 w.r.t. panel k
      LU( A(:,k), A(:,1:k-1) );
      #pragma omp section
      // Update panels k+1:n_b w.r.t. panel k
      TU( A(:,k), A(:,k+1:n_b) );
  } }
}
```

Listing 3: Simplified routine for DMFI, with hybrid parallelism extracted via OpenMP sections plus invocations to multi-threaded BLAS from within LU and TU.

on the underlying runtime to detect task dependencies dynamically, at execution time.

3. NUMA-aware parallel dense matrix factorizations and inversion

The parallel approaches described in Section 2 in general hit a memory wall when a thread running in a particular core has to access data that lies in a different NUMA Node. As a result, the data must be transferred over the NUMA connection, at a slow rate due to contention and data movement across the network, increasing the time cost of the global computation.

In this section we advocate a programming approach for DMFI, with a high level of abstraction, that nonetheless is aware of the underlying NUMA memory organization. In the following we will consider a generic NUMA architecture composed by a set of NUMA nodes (NN); all the explanations hereafter are valid for any number of NNs, and independent of the “distance” between the core and the data (viewed as an abstraction of the cost of moving to a specific core a piece of data from a specific memory address that can be *local* or *remote* to the core, depending on the NN to which the core belongs to and where the memory is allocated to).

3.1. Strategies for NUMA-aware DMFI

Ensuring that a thread mostly access data that is in the *local* NN requires an explicit control of two key aspects: *(i)* the data allocation policy across NNs; and *(ii)* the workload distributions across threads/cores within the multi-core system. We next comment how to deal with them depending on the parallelization approach.

On the one hand, when parallelism is extracted only at the loop level, as is the case of LAPACK when linked with a multi-threaded version of the BLAS, the threads should majorly execute those loop iterations that operate with local data, which would require a complete rewrite in order to obtain a NUMA-aware BLAS. Although it is possible to use NUMA-aware capabilities within the BLAS (e.g., BLIS [40] includes NUMA-aware multi-threaded implementations), not all BLAS realizations support this feature.

On the other hand, when parallelism is exploited at the task level, the worker threads should mostly run those tasks that involve only local data. This variant requires the use of NUMA-aware task schedulers, in which metrics for estimating the memory distance are integrated within the *Task Dependency Graph* (TDG) in order to expose and exploit NUMA locality. While this has been prototyped by some task schedulers (e.g., OmpSs [3,8,37]), NUMA support in this type of system software is not widely extended and requires a complex development and runtime logic in order to be effective.

In summary, for high performance, both approaches require a careful orchestration of both data and workload distributions, presenting many similarities with distributed parallel programming, including the code complexity of the latter. A hybrid solution, in which parallelism is extracted simultaneously at the loop and task levels, offers a simpler mechanism to map DMFI to NUMA architectures, provided the three following techniques are properly combined in order to tackle the intricacies of NUMA systems:

1. *Thread placement and binding to cores*, to ensure that locality in data references is consistent across the complete operations;
2. *Data partitioning*, so that data structures are correctly scattered across NNs and hence accessed locally by the appropriate threads; and
3. *NUMA-aware parallel algorithms* to take into account both workload and data distribution that favors NUMA-aware data accesses.

In the following, we describe a common NUMA-aware framework for DMFI. For this purpose, we revisit some of the ideas introduced in previous works for hybrid parallel programming for DMFI on shared-memory [12] and distributed-memory architectures [7], while adapting and extending them to be efficiently mapped to modern NUMA architectures. The goal is hence to obtain NUMA-aware realizations for DMFI routines that fulfill the following considerations:

1. Matrices are logically and physically partitioned into *domains* that, on a NUMA architecture, should be distributed and reside on different NNs throughout the complete operation. Note that the *domain* is a purely software artifact (and hence, they should be explicitly addressed from the codes), while the *NN* reflects a hardware concept. Intuitively, the ultimate goal is to establish generic and simple mechanisms to *expose the domains in code*, and to then *map the exposed domains to the NNs* at runtime.
2. At runtime, threads are deployed following a hierarchical structure, with a first level of parallelism in which a single thread is bound to a specific NN (concretely, to those cores that are *close* to that NN); and a second level of parallelism which inherits the thread-to-core assignment to execute each task in parallel.
3. A constant thread-to-core (and to-memory) assignment is in place throughout the complete computation, applying the *owner-computes* rule on the corresponding *domain* [20].

The baseline hybrid algorithm in Listing 3 needs to be slightly modified to ensure a fine-grain, explicit control of the thread-to-task assignment and that the work distribution is consistent through the loop iterations of the DMFI. Listing 4 shows the proposed modifications to the implementation, showing a generic DMFI that replaces the use of OpenMP `sections` by an explicit thread identifier query and work distribution. This will be our baseline for the discussion on NUMA-aware DMFI executions. In this specific example, PF is always assigned to Thread 1, while LU, TU are respectively assigned to Threads 1, 2 throughout the whole computation. (For consistency with the notation for matrix indices, the threads identifiers are numbered starting at 1.)

3.2. Data partitioning into multiple domains

In our solution we adopt a cyclic block-column distribution of the matrix, generating multi-domain logical and physical representations of the original matrix. The number of domains is configurable; for example, decomposing a matrix A into four domains means that the matrix panels $A(:,1)$, $A(:,5)$, $A(:,9)$, ... are

```
void DMFI_NUMA_HP( matrix A, int n_b )
{
  for ( int k = 1; k <= n_b; k++ ) {
    #pragma omp parallel num_threads(2)
    {
      int thid = omp_get_thread_num() + 1;

      if( thid == 1 )
        // Factorize panel k
        PF( A(:,k) );
    }
    #pragma omp parallel num_threads(2)
    {
      int thid = omp_get_thread_num() + 1;

      if( thid == 1 )
        // Update panels 1:k−1 w.r.t. panel k
        LU( A(:,k), A(:,1:k−1) );

      if( thid == 2 )
        // Update panels k+1:n_b w.r.t. panel k
        TU( A(:,k), A(:,k+1:n_b) );
    }
  }
}
```

Listing 4: Simplified routine for DMFI, with hybrid parallelism extracted via OpenMP plus invocations to multi-threaded BLAS from within LU and TU, including a fine-grain control of thread-to-task assignment.

```
void Copy_NUMA_HP_4D( matrix A,
                      matrix D1, D2, D3, D4,
                      int n_b )
{
 int k_d;

 #pragma omp parallel num_threads(4) private(k_d)
 {
   int thid = omp_get_thread_num() + 1;

   for ( int k = thid; k <= n_b; k+=4 ) {
     k_d = (k−1) / 4 + 1;
     switch( thid ) {
        case 1: // Copy to D1
          Copy( A(:,k), D1(:,k_d) ); break;
        case 2: // Copy to D2
          Copy( A(:,k), D2(:,k_d) ); break;
        case 3: // Copy to D3
          Copy( A(:,k), D3(:,k_d) ); break;
        case 4: // Copy to D4
          Copy( A(:,k), D4(:,k_d) ); break;
     }
   }
 }
}
```

Listing 5: Simplified parallel routine for copying a matrix A into four domains, including a fine control of thread-to-task assignment.

mapped to panels $D1(:,1)$, $D1(:,2)$, $D1(:,3)$, ... of a local matrix $D1$. Panels $A(:,2)$, $A(:,6)$, $A(:,10)$, ... are mapped to $D2(:,1)$, $D2(:,2)$, $D2(:,3)$, ... in a different local matrix $D2$; and so on.

Listing 5 provides a realization of a parallel routine with fine-grain task-to-thread assignment for the copy of a matrix A into four domains: $D1$, $D2$, $D3$, $D4$. A multi-domain matrix generation routine would follow a similar scheme to the copy routine. This type of parallel manipulation of multiple logical and physical domains will be leveraged in Section 3.4 to attain a physical distribution of domains across NNs. At this point we note that, even though a 2D cyclic partitioning in general provides higher parallel scalability than a simpler 1D distribution, the number of NNs in current NUMA architectures is moderate, in the range 2–16, and therefore a cyclic column-block may be sufficient from the perspective of parallel performance.

3.3. Multi-domain hybrid parallel algorithm for DMFI

Listing 6 generalizes the hybrid parallel code in Listing 3 to operate on an input matrix partitioned into four domains (the same ideas apply to other number of domains). In this example, the code deploys a first level of parallelism in which four threads (in the following, we will refer to each thread in the first-level as a *Way*) execute the main DMFI building blocks in an order established by the programmer.

The code remains quite simple and structured: at each iteration, the current panel is factorized by a distinct *Way*, and the column-blocks to the left and right in the four domains are then updated with respect to the factorized panel. From the programming point of view, most of the implementation details remain hidden inside the kernels PF, TU, LU, which contain exactly the same code as in the case without support for multiple domains. The programming burden is thus reduced to determining the correct starting block indices when accessing the contents of $D1$, $D2$, $D3$, $D4$.

The multi-domain algorithm represents a first effort toward *work assignment* to Ways as, for each iteration, each domain is updated (written) by the same thread, following the *owner-computes* rule. This work-assignment is not mandatory for correctness, but will be basic for the NUMA-aware realizations discussed next.

3.4. NUMA-aware multi-domain DMFI executions

A NUMA-aware execution of the multi-domain algorithm depicted in Listing 6 should perform a fine-grain control on thread placement and work assignment in order to increase the number of local memory accesses and control the distribution of threads to NNs. Specifically, our proposal combines the use of OpenMP runtime environment variables and the multi-domain DMFI algorithms on four intimately related dimensions yielding a methodology for a NUMA-aware execution:

1. *Place definition.* We employ the `OMP_PLACES` environment variable to match the physical NUMA topology of the underlying architecture with the logical thread topology considered by the OpenMP runtime. Hence, for example, on a 2-socket, 128-core machine with two NNs (one per socket), comprising 64 consecutive cores each, this definition would set

   ```
   OMP_PLACES={0:64:1},{64:64:1}
   ```

   whereas a configuration of the same machine to define two NNs per socket (comprising 32 consecutive cores each) would employ

   ```
   OMP_PLACES={0:32:1}, {32:32:1},\\
               {64:32:1},{96:32:1}
   ```

2. *Thread-to-core binding.* Consider a NUMA-aware execution with a total of `NT` threads mapped to 4 NNs. We should then deploy 4 Ways, each one creating $NT/4$ threads per BLAS invocation and in charge of updating one domain. In a NUMA-aware execution, the Ways need to be scattered across OpenMP places (following a `spread` OpenMP binding policy at the first thread level), and threads deployed within BLAS calls should inherit the thread-to-place binding of each original Way (following a `close` OpenMP binding policy on the second thread level). For this purpose, the value of the `OMP_NUM_THREADS` and `OMP_PROC_BIND` environment variables should be fixed as

   ```
   OMP_NUM_THREADS="4,$NT_PER_BLAS_CALL"
   OMP_PROC_BIND="spread,close"
   ```

```
void DMFI_NUMA_HP_4D( matrix D1, D2, D3, D4,
                       int n_b )
{
 int k_d, k_m;

 for ( k = 1; k <= n_b; k++ ) {
  k_d = (k−1) / 4 + 1;
  k_m = (k−1) % 4 + 1;
  switch ( k_m ) {
    case 1:
      #pragma omp parallel num_threads(4)
      {
        int thid = omp_get_thread_num() + 1;

        if( thid == 1 ) // Fact. panel k in D1
          PF( D1(:,k_d) );
      }
      #pragma omp parallel num_threads(4)
      {
        int thid = omp_get_thread_num() + 1;

        // Update of panels 1:k−1 w.r.t. panel k
        // omitted for brevity
        // Update panels k+1:n_b w.r.t. panel k

        switch( thid ) {
        case 1: // Update D1 w.r.t D1.
          TU( D1(:,k_d), D1(:,k_d+1:end) ); break;
        case 2: // Update D2 w.r.t D1.
          TU( D1(:,k_d), D2(:,k_d:end) ); break;
        case 3: // Update D3 w.r.t D1.
          TU( D1(:,k_d), D3(:,k_d:end) ); break;
        case 4: // Update D4 w.r.t D1.
          TU( D1(:,k_d), D4(:,k_d:end) ); break;
      } }
      break;
    case 2:
      #pragma omp parallel num_threads(4)
      {
        int thid = omp_get_thread_num() + 1;

        if( thid == 2 ) // Fact. panel k in D2
          PF( D2(:,k_d) );
      }
      #pragma omp parallel num_threads(4)
      {
        int thid = omp_get_thread_num() + 1;

        // Update of panels 1:k−1 w.r.t. panel k
        // omitted for brevity
        // Update panels k+1:n_b w.r.t. panel k

        switch( tid ) {
        case 1: // Update D1 w.r.t D2
          TU( D2(:,k_d), D1(:,k_d+1:end) ); break;
        case 2: // Update D2 w.r.t D2
          TU( D2(:,k_d), D2(:,k_d+1:end) ); break;
        case 3: // Update D3 w.r.t D2
          TU( D2(:,k_d), D3(:,k_d:end) ); break;
        case 4: // Update D4 w.r.t D2
          TU( D2(:,k_d), D4(:,k_d:end) ); break;
      } }
      break;
    case 3:
     // Code ommitted for brevity
    case 4:
     // Code ommitted for brevity
 } }
}
```

Listing 6: Simplified routine for DFMI using a multi-domain scheme with hybrid parallelism extracted via OpenMP parallel regions plus invocations to multi-threaded BLAS from within LU and TU.

The use of this specific combination of hierarchical parallelism, thread-to-core binding, and definition of places yields a proper distribution of threads across NNs throughout the computation, and can be adapted (departing from the same multi-domain DMFI code) to different NUMA topologies without code modifications, provided the number of domains exposed in the code matches the target number of NNs.

3. *Workload distribution.* Considering that the OpenMP implementation maintains the thread identification across parallel regions, the workload control in Listing 6 would suffice to assure that each deployed Way and BLAS threads spawned within it fulfill the *owner-compute* rule throughout the complete operation.
4. *Thread-to-data affinity.* Finally, a mechanism to ensure that the data accesses from within a Way or from the internal BLAS threads are local, so that the updates are performed within the local NN, is mandatory to prevent remote memory accesses. For this condition to hold, each Way in the domain creation depicted in Listing 5 should be mapped to a core bound to a different NN, in order to exploit the *first-touch* page allocation policy present in modern operating systems.[2] This mapping, however, is already active provided the aforementioned *(1)* place definition, *(2)* thread-to-core mapping, and *(3)* workload distribution are used, and hence a NUMA-aware implementation (with local domain updates) will be consistently used.

3.5. Generalization to any number of NUMA nodes

Although the multi-domain algorithms are easy to derive, their implementation can be tedious when the number of domains is large. Hence, generalizing the implementation to be independent of the number of domains becomes very useful for code and performance portability across NUMA architectures as it requires minimal (or no) code modifications. An excerpt of the solution is given in Listing 7, for a configuration with n_d NUMA domains. Note that we employ an array of domain descriptors as the source of the DMFI, with as many elements as domains conform the operation; this structure is the multi-domain representation of the matrix A, and in practice could be defined as an object that abstracts away the intricacies of managing a multi-domain representation of the matrix. The listing also includes an excerpt of a generic implementation of the routine that copies the contents of a plain matrix A to the n_d domains. With these two generic implementations, the development of multi-domain codes for DMFI becomes simple, and enables the extensive performance evaluation for different NUMA topologies in Section 4.

4. Experimental results

The experimental evaluation in this section pursues three main objectives. First, to demonstrate that our NUMA-aware shared-memory approach toward obtaining efficient parallel DMFI implementations is valid for systems with a variable number of NNs, and to illustrate the implications of a proper domain number selection in relation with the number of NNs in the system; these results are given in Sections 4.2 to 4.4. Second, to compare the attained performance with message-passing implementations for DFMI (ScaLAPACK and SLATE); as reported in Section 4.5. Third, to demonstrate that the proposed approach is portable across a number of routines (LU, QR and Inversion) and new-generation NUMA architectures; this is done in Section 4.6. We employ the GFLOPS rate (billions of floating point operations per second) as the main performance metric in all subsequent experiments, and we use double-precision arithmetic. The reported results are the best from a large number of repetitions per experiment in order to reduce variability. As the two target architectures are quite novel from

[2] An execution of the factorization executable using `numactl -localalloc` would be necessary if the *first-touch* policy is not in place.

```
// Generic multi-domain DMFI.
void DMFI_NUMA_HP_generic( matrix $D[n_d]$,
                           int $n_b$, int $n_d$ )
{
 int $k_d$, $k_m$;

 for ( int $k$ = 1; $k$ <= $n_b$; $k$++ ) {
  $k_d$ = $(k-1)$ / $n_d$ + 1;
  $k_m$ = $(k-1)$ % $n_d$ + 1;

  #pragma omp parallel num_threads($n_d$)
  {
    int thid = omp_get_thread_num() + 1;

    // Factorize panel $k$ in Domain $k_m$
    if( thid == $k_m$ )
      PF( $D[k_m](:,k_d)$ );
  }

  #pragma omp parallel num_threads($n_d$)
  {
    int thid = omp_get_thread_num() + 1;

    // Update of panels $1{:}k-1$ w.r.t. panel $k$
    // omitted for brevity

    // Update panels $k+1{:}n_b$ w.r.t. panel $k$
    // Each thread updates the proper domain
    int $k_i$ = ($k_m$ <= thid) ? $k_d$ + 1 : $k_d$;
    TU( $D[k_m](:,k_d)$, $D[\text{thid}](:,k_i:end)$ );
  }
}

// Generic copy of matrix A into domains
void Copy_NUMA_HP_generic( matrix $A$, matrix $D[n_d]$,
                           int $n_b$, int $n_d$ )
{
 int $k_d$;

 #pragma omp parallel num_threads($n_d$) private($k_d$)
 {
  int thid = omp_get_thread_num() + 1;

  for ( int $k$ = thid; $k$ <= $n_b$; $k$+=$n_d$ ) {
   $k_d$ = $(k-1)$ / $n_d$ + 1;
   // Copy to the corresponding Domain
   Copy( $A(:,k)$, $D[\text{thid}](:,k_d)$);
  }
 }
}
```

Listing 7: Simplified routine for DFMI, generalized for n_d NUMA domains, and with NUMA-aware hybrid parallelism extracted via OpenMP parallel regions plus invocations of multi-threaded BLAS from within LU and TU.

the perspective of their NUMA characteristics, Section 4.1 provides their detailed description.

All the performance results in the following exclusively consider the cost of the factorizations/inversion while the time devoted to create and/or distribute data across domains is not included. The reason is two-fold: First, a sequential implementation of the routine `Copy_NUMA_HP_generic` in Listing 7 entails a time penalty between 6 and 15% for problems of relatively large dimension; in this line, we can expect a significantly smaller impact when using a parallel copy routine to saturate the memory bandwidth. Second, in a natural scenario, these routines maintain the matrix operand distributed across the NUMA nodes prior and after the invocation of the corresponding routines, hence making it unnecessary (an inefficient) to perform a data distribution to/from NUMA domains every time a DMFI routine is executed.

4.1. Experimental setup

4.1.1. Rome

Rome is a dual-socket multi-core server equipped with two AMD EPYC 7742 processors with 64 cores each, configured in our setup to run at 2.25 GHz. The cache hierarchy per socket comprises 4 Mbytes of L1 cache (32 Kbytes per core), 32 Mbytes of L2 cache (512 Kbytes per core), and 256 Mbytes of L3 cache divided into 16 modules of 16 Mbytes, local to each cluster of 4 cores. The server features 512 Gbytes of DDR4 (3200 MHz) RAM memory. The memory controller supports up to 8 memory channels, for an aggregated peak theoretical bandwidth of 204 Gbps. The chip is logically divided into four quadrants, each one comprising 16 cores and associated with two memory channels.

The basic compute unit (core) in Rome implements the Zen2 micro-architecture, a superscalar design supporting vector instructions of up to 256 bits. Cores are grouped within the Rome SoC (System on Chip) into Core-Complexes (CCX). Each CCX in Rome is composed of 4 cores[3] sharing 16 Mbytes of L3 cache. CCXs are grouped in pairs to conform a CCD (Core/Cache Die), which are in turn logically and physically distributed into 4 quadrants, with 2 CCDs per quadrant. CCDs are actually the basic chiplet (die) unit within the processor, and scalability across products in the same family is attained by means of CCD replication. In Rome, the 8 CCDs (chiplets) are interconnected via a ninth I/O die, which manages intra-socket (die-to-die) and inter-socket communication via the so-called *Infinity* fabric. A pair of memory channels, each one bound to two RAM DIMMs, is bound to each quadrant. Fig. 2 depicts the global structure of the Rome processor employed in our tests, and a detail of the CCX structure.

Given this die, cache hierarchy and memory topology, the Rome SoC offers the possibility of establishing (via BIOS setup) several *Nodes Per Socket (NPS)* configurations, which ultimately impact the NUMA topology visible for users and the OS, in terms of logic distance, modifying the System Locality Distance Information Table (SLIT)[4] [41] as well as the effective bandwidth that can be attained by means of assignment of memory channels to specific cores in the SoC. Specifically, in Rome, the following configurations for NPS are available:

NPS1 interleaves the eight channels in the socket for their use by all cores in the processor; hence, the complete socket is configured as a single NN (typical in common multi-socket NUMA setups). In our dual-socket setup, NPS1 yields 2 NNs.

NPS2 interleaves the four channels in each half of the chip for exclusive use of the corresponding cores; hence, each half is configured as an NN. In our dual-socket setup, NPS2 yields 4 NNs.

NPS4 interleaves the two channels in each quadrant for exclusive use of the corresponding cores; hence, each quadrant is configured as an NN. In our dual-socket setup, NPS4 yields 8 NNs.

LLCasNUMA considers each CCX as a single NN, offering 16 NNs per socket; the utility of this setup lies in a complete resource isolation (including LLC) that is desirable in some multi-tenant setups in datacenters. In our dual-socket setup, LLCasNUMA yields 32 NNs.

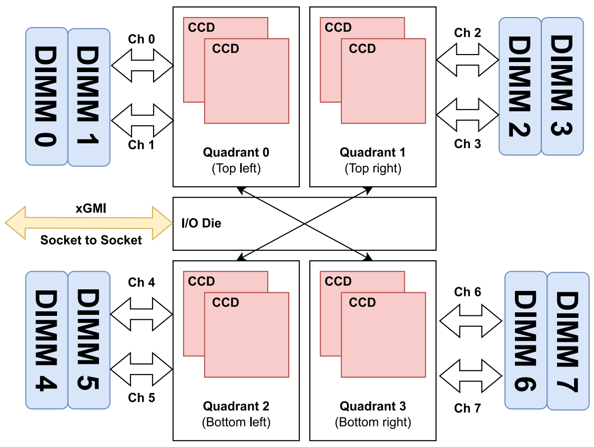
From the software side, Rome runs a Linux 5.10 machine using the GNU Compiler suite version 8.3 and AOCL version 3.1.0.

4.1.2. Kunpeng

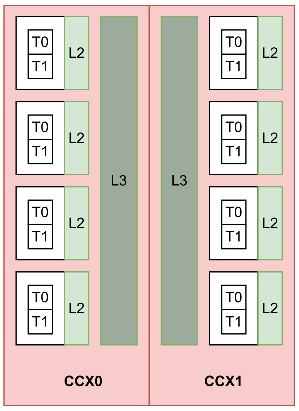
Kunpeng is a dual-socket multi-core server equipped with two Huawei Kunpeng 920 processors with 48 cores each, configured in our setup to run at 2.6 GHz. The cache hierarchy per socket comprises 3 Mbytes of L1 cache (64 Kbytes per core), 24 Mbytes of L2 cache (512 Kbytes per core), and 48 Mbytes of L3 cache (1

[3] 2-way SMT per core is supported, though in our experiments this capability was disabled.

[4] Typically queried from the user space via `numactl -H`.



(a) Rome socket architecture.

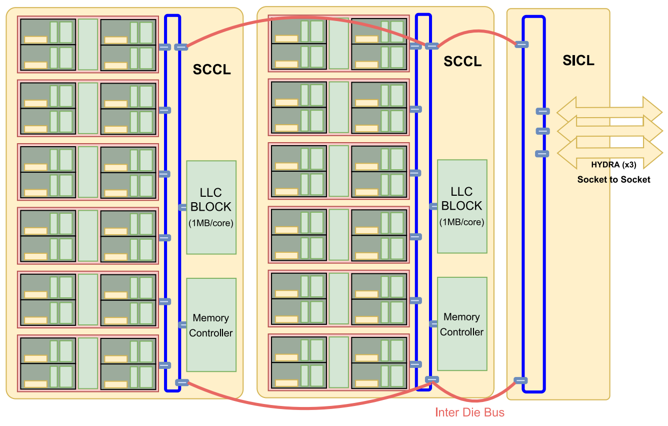


(b) Rome CCD architecture.

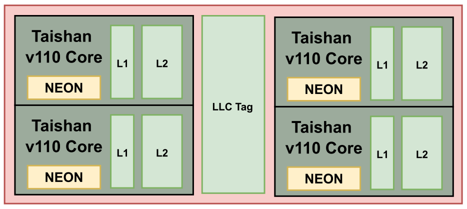
Fig. 2. Memory hierarchy and core layout of Rome.

Table 1
Theoretical memory bandwidth and latency for a multi-socket Kunpeng 920 architecture (source: [48]).

	Bandwidth (GB/s)	Latency (ms)
Core (L1)	6000	<2
L2	6000	<4
L3	3000	<15
Inter-die	400	<30
Main memory	200	<90 – 110
Inter-socket	50-90	<230



(a) Kunpeng socket architecture.



(b) Kunpeng CCL architecture.

Fig. 3. Memory hierarchy and core layout of Kunpeng.

Mbyte per core). The server features 256 Gbytes of DDR4 (2933 MHz) RAM memory. Similar to Rome, the memory controller supports up to 8 memory channels, for an aggregated peak theoretical bandwidth of 204 Gbps.

The basic computing unit (core) within Kunpeng is the TaiShan v110 core, an ad-hoc implementation of the ARM v8.1 (with selected extensions from the ARM v8.2 specification) and support for 128-bit NEON vector instructions. Cores are grouped within the Kunpeng SoC into CPU clusters (CCLs), each comprising 4 TaiShan v110 cores. At a higher level, the chip is composed by three dies (chiplets), namely: two Super CPU Clusters (SCCL) compute dies and one Super IO Cluster (SICL). Each SCCL is composed by a variable number of CCLs (in our case 6 CCLs, for a total amount of 24 cores per SCCL), a memory controller, and an LLC cache block. Core scalability within the SCCL chiplet is based on replication of CCLs within it. Communication across cores in an SCCL is implemented via a ring topology, with the SICL die in charge of managing inter-die communication. The SICL also includes three Hydra links for inter-socket communication, with an aggregated peak bandwidth of 90 Gbytes per second. Table 1 offers a comparison of the memory access characteristics at the distinct levels on Kunpeng. Especially interesting for us are the two main bandwidth/latency gaps for inter-die and inter-socket communications, which will heavily impact the final performance on remote memory accesses that need to traverse those levels.

Fig. 3 depicts the global structure of the Kunpeng processor employed in our tests, and a detail of the CCL structure. In this case, NUMA effects within the server can appear across chips, and across the two SCCLs (with proprietary LLCs) within each chip, for a total of 4 NNs. We have not investigated in possible modifications of the NUMA topology as we did in Rome.

From the software side, our experiments were deployed on a Linux 5.4.0 machine using the GNU Compiler suite version 10.1 and ARMPL (ARM Performance Libraries) version 21.1.

4.2. Impact of NPS on performance

The goal of the first round of experiments is three-fold: First, to compare the performance improvements of our multi-domain DMFI implementations with classic single-domain (LAPACK-style) DMFI implementations; second, to evaluate the potential improvements in performance of the multi-domain NUMA-aware DMFI executions compared with their NUMA-oblivious counterparts, in which the OS is in charge of data distribution and workload assignment; and third, to assess how the number of domains selected from the library are directly related with the performance improvement as the number of NPS (and hence the number of NNs in the whole system) is increased.

In this case, we select a specific DMFI (the QR factorization), and Rome as the target architecture. Similar qualitative results were observed for the rest of DMFI (LU and matrix inversion). The reason for selecting Rome was its ability to experiment with a larger variety of NPS configurations, which illustrates the flexibility of our solution.

Fig. 4 reports the performance results on three different configurations of the server: NPS1 (top plot), NPS2 (middle), and NPS4 (bottom) for increasing problem dimensions. Each group of bars reports the performance for a specific problem dimension, for different library setups:

- *1 domain, no NUMA* (dark blue bar): DMFI with a single data domain, extracting parallelism at only one level (within BLAS calls), and delegating the control of thread-to-core affinity and data allocation to the OS.
- *X domains, no NUMA* (light version of the corresponding color): DMFI with X data domains, extracting parallelism at two levels, and delegating the control of thread-to-core affinity and data allocation to the OS (as explained in Section 3.3).
- *X domains, NUMA* (dark version of the corresponding color): DMFI with X data domains, extracting parallelism at two levels, and explicitly controlling thread-to-core affinity and data allocation (as explained in Section 3.4).

For the multi-domain experiments, we execute the DMFI codes for 2, 4, 8 and 16 domains. In all cases, our executions comprise the 128 cores, that is, the complete (dual-socket) machine.

A number of conclusions can be extracted from the analysis of the results:

- In general, the use of our multi-domain NUMA-aware DMFI implementation outperforms the classic single-domain approach. This difference in performance becomes more visible as the dimension of the matrices grows, and as the number of NPS is increased, as in such scenario(s), it is more challenging for the OS to find an optimal allocation scheme for data and/or a proper thread-to-data binding.
- For each selection of the number of domains, comparing the NUMA-aware implementations with their counterparts in which the OS manages data affinity for each NUMA setup, the benefits in terms of performance are evident in all cases. This fact reveals the importance of exposing multiple domains within the code as well as to manually control, at execution time, the affinity of threads to cores and hence to data domains.
- Regarding the relationship between the number of NNs and the number of domains, the performance boost appears in all cases exactly when the number of domains matches the number of NNs (that is, two domains for NPS1, four domains for NPS2, and eight domains for NPS4). This establishes the minimum number of domains that should be configured according to the underlying NUMA setup and, more importantly, demonstrates the necessity of a flexible shared-memory DMFI library implementation that can accommodate any number of NNs for performance portability.
- According to AMD's documentation, NPS4 is the recommended NUMA setup in order to attain optimal aggregated memory bandwidth. While this usually holds for multi-application environments in which applications are confined and mapped to different NNs, our observations reveal that this is not the case when there is only one application which spans across multiple NNs. Here, when respecting a correct distribution of domains to NNs, the best configuration is that which reduces the number of NNs (NPS1), with performance dropping progressively as the number of nodes for the same experiment conditions increases.

Additionally, Fig. 5 (left) reports the performance for the LL-CasNUMA configuration, in which 16 NNs are set per socket. In this case, the results correspond to an execution on a single socket (64 cores). Our goal here is to demonstrate that a NUMA-aware

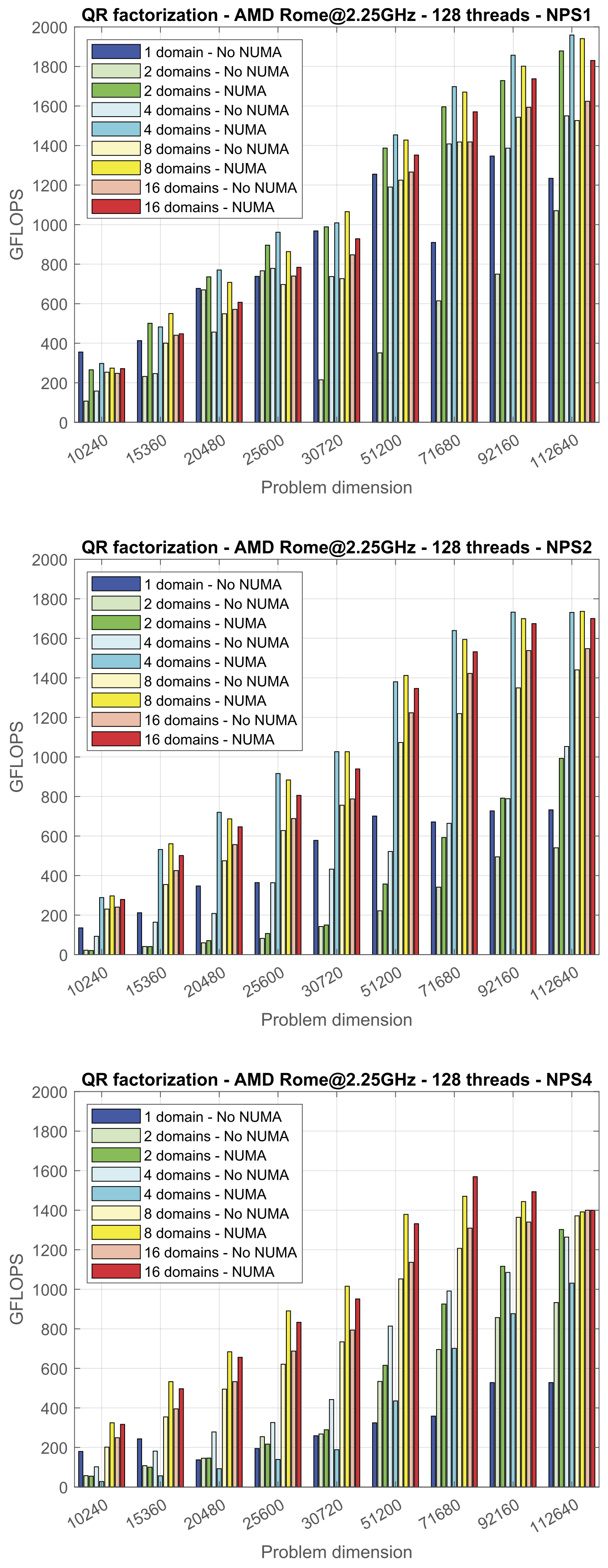
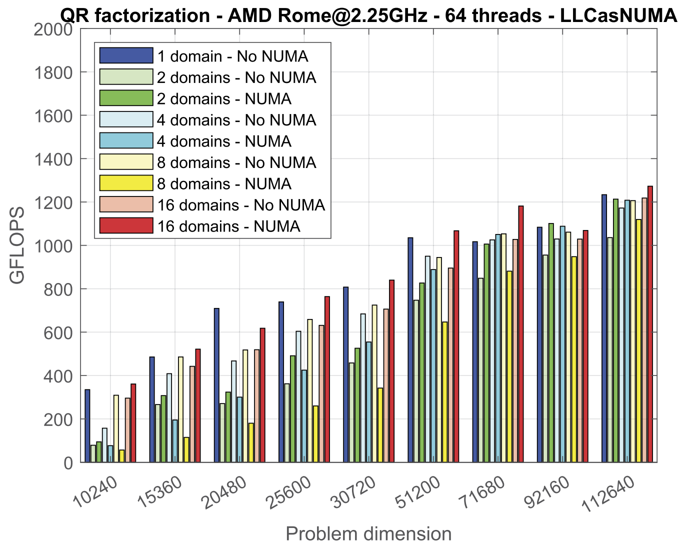


Fig. 4. Comparative performance results for the QR factorization under different NPS configurations on Rome.



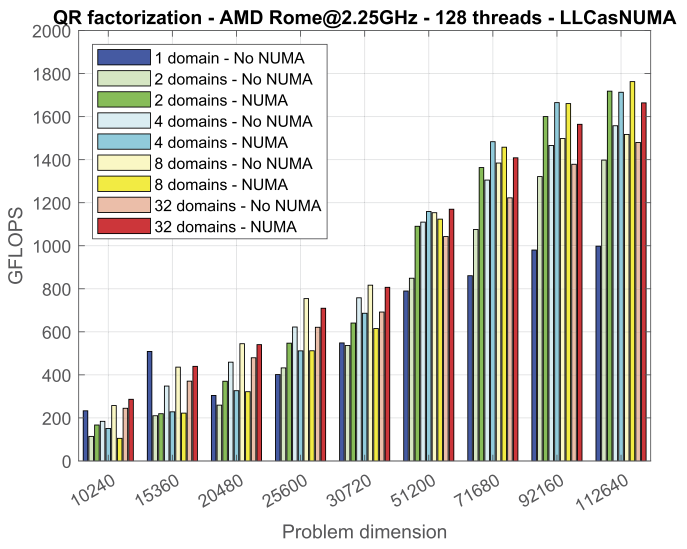


Fig. 5. Comparative performance results for the QR factorization the LLCasNUMA configuration on Rome. Left: using 1 socket and up to 16 domains; Right: using 2 sockets and up to 32 domains.

execution with a number of domains which matches the number of NNs is also profitable when employing a single socket. In this case, the NUMA-aware execution with 16 domains (dark-red bar) again delivers the highest performance. However, the NUMA effects on performance are less evident in these cases, and actually, for this type of regular codes (from the perspective of memory accesses), the basic single-domain codes (dark-blue bar) are rather competitive. Our guess is that the management of homogeneous distances across all NNs within a single socket helps the OS to map the threads close to the corresponding NN, and also that potential performance (latency or bandwidth) penalties associated with remote data accesses are not as dramatic as in multi-socket setups. To close this first round of experiments, Fig. 5 (right) reports equivalent results using the LLCasNUMA setup on both sockets of Rome. Again, the fine-grain NUMA node distribution in this setup blurs the benefits of increasing the number of domains in the NUMA-aware codes. Here, the largest leap in performance arises when moving to two domains, as the logical distance between NUMA nodes across both sockets is, proportionally, the main source of inefficiency in the NUMA-oblivious implementation. In other words, LLCasNUMA and NPS1 are the most similar NUMA setups available in the machine for this type of compute-intensive implementation. Note that, differently from the single-socket setup, the use of a LAPACK-like 1-domain implementation introduces a significant performance penalty.

4.3. Performance counters

Hardware performance counters can help to explain the difference in performance between NUMA-aware and NUMA-oblivious executions observed in the previous section. PMUs (Performance Monitoring Units) in modern NUMA architectures provide detailed information about the number of accesses to local and remote DRAM. In order to support the results reported in the previous section, we leverage the PMU in Rome in order to gather events that correspond to remote DRAM accesses; these events are provided by the Core Performance Monitor Counters (PMC) of the PMU; see [4] (section 2.3.2, LS Events).

Fig. 6 reports an execution trace comparing the distribution of the remote DRAM accesses for two different executions of the same multi-domain QR factorization, using NUMA-aware (blue line) and NUMA-oblivious (brown line) execution setups. The specific event from the PMC is 0x043 (*Data Cache Refills from System*),

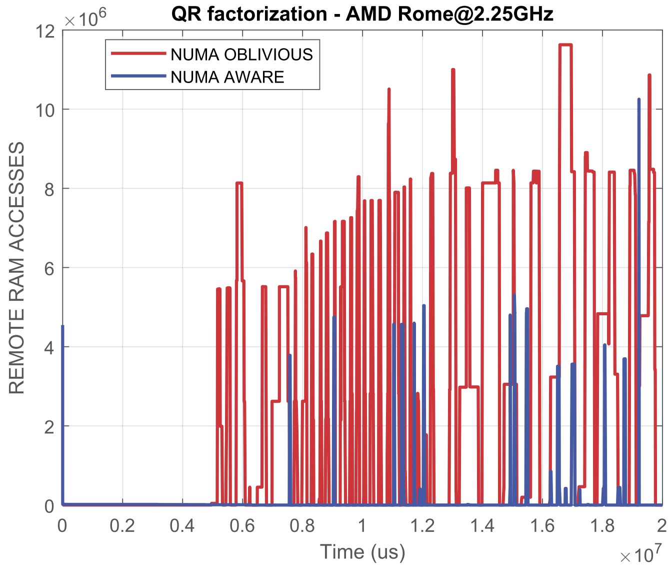


Fig. 6. Comparative timeline of the remote RAM accesses for a QR factorization ($m = n = 1,024$, $b = 256$) in Rome for a NUMA-aware and a NUMA-oblivious execution.

specifically activating the event `LS_MABRESP_RMT_DRAM` (*DRAM accesses from another die*). NPS1 and two domains were used for the experiment, even though similar qualitative results were observed for other configurations. The codes were instrumented using PAPI 6.0.0. The timeline in the Figure delivers a much higher rate of remote memory accesses in the NUMA-oblivious execution compared with that of its NUMA-aware counterpart, which explains the difference in performance, and hence the necessity of a combined multi-domain code for DMFI and a NUMA-aware execution.

4.4. Performance portability

The goal of this section is to provide evidence on the performance portability of our NUMA-aware codes for DMFI across the different NPS configurations in Rome, and expose how the performance degrades as the number of NNs is increased. To accomplish the study, we conduct a comparative study of the qualitative and quantitative performance of our solution, taking the General Matrix-Matrix Multiplication (GEMM hereafter) as a base-

Table 2
Performance portability study of GEMM and QR across different NPS configurations on Rome, using 128 threads.

	NUMA-aware				NUMA-oblivious					
	GEMM		QR (multi-domain)		GEMM		QR (1 domain)		QR (multi-domain)	
	Max. GFLOPS	Relative to NPS1	Max. GFLOPS	Relative to NPS1	Max. GFLOPS	Relative to NPS1	Max. GFLOPS	Relative to NPS1	Max. GFLOPS	Relative to NPS1
NPS1	3,237	–	1,958	–	2,734	–	1,346	–	1,550	–
NPS2	2,924	90.3%	1,732	88.4%	1,788	65.3%	732	54.3%	1,440	92.9%
NPS4	1,723	53.2%	1,470	75.1%	1,652	60.4%	527	39.2%	1,371	88.5%
LLCasNUMA	3,045	94.1%	1,762	89.9%	2,506	91.6%	998	74.1%	1,517	97.8%

line. GEMM usually serves as a realistic benchmark to evaluate the maximum attainable performance of an architecture for compute-intensive applications. Hence, this study also aims at illustrating the gap (in terms of raw performance) between the DMFI and GEMM for each configuration.

Table 2 reports the results of the study using the complete Rome machine (128 cores). The values there include the maximum performance (in terms of GFLOPS) attained for the QR factorization and GEMM, and the relative performance compared with the best NUMA configuration (in our observations, NPS1) for each routine. For clarity, we divide the following discussion into two parts, targeting NUMA-aware and NUMA-oblivious executions.

4.4.1. NUMA-aware implementations

The left part of Table 2 reports the performance portability results for the best NUMA-aware execution of our multi-domain codes as reported in Figs. 4 and 5 (right). For GEMM, we have conducted an evaluation of AOCL considering a NUMA-aware distribution of threads, following the directives presented in [40], in which the number of thread groups in the outer AOCL loop (referred as `JC` in that work) is manually set to match the number of NNs in the NPS setup (e.g., 2 for NPS1, 4 for NPS2, and so on). This, in turn, yields a NUMA-aware AOCL implementation for GEMM and hence a fair baseline for the comparison.

The results show that the relative performance compared with NPS1 follows a similar trend for our DMFI and GEMM: The performance degrades in a similar ratio as the number of NNs increases, and offers a decent degradation for LLCasNUMA. As mentioned in Section 4.2, the NUMA setup for LLCasNUMA yields logical distances similar to those in NPS1, hence the higher comparative performance in this case (94.1% for GEMM and 89.9% for the QR factorization compared with NPS1).

Regarding the maximum observed performance, the QR factorization attains rates between a 60% (for NPS1) and 85% (for NPS4) compared with those in GEMM. This reduction in performance is common in DMFI, and is explained by the bottleneck introduced by the panel factorization; all in all, our maximum performance is comparable with that in alternative implementations such as ScaLAPACK or SLATE. (To be shown in Section 4.5.)

4.4.2. NUMA-oblivious implementations

The right part of Table 2 reports the results for the best NUMA-oblivious execution of our multi-domain codes as reported in Figs. 4 and 5 (right). In this case, for GEMM we just set the total number of threads for the AOCL execution (128), and delegate the actual thread distribution to the library. Hence, we can consider this GEMM instance as NUMA-oblivious. The table also includes the execution of a 1-domain QR factorization linked with AOCL (configured using 128 threads), that mimics the implementation of the LAPACK's `dgeqrf` routine and serves here as a baseline for the performance comparison with a NUMA-oblivious DMFI.

The conclusions in this case are qualitatively similar to those observed for the NUMA-aware case: the performance experiments a similar degradation with the NPS variation. In terms of absolute performance comparison with GEMM, the difference with our DMFI solution is similar to that in the NUMA-aware experiments. In both cases, the NUMA-oblivious executions suffer a similar performance reduction w.r.t. their NUMA-aware counterparts. However, the degradation is dramatically milder for our multi-domain DMFI codes (even on a NUMA-oblivious execution) than those for GEMM and for LAPACK's `dgeqrf`; this observation reinforces the relevance of exposing multiple domains from the code, which eases the task of NUMA-aware thread binding for the operating system, enhancing performance portability.

4.5. Comparison with message-passing libraries

In order to put in context the raw performance of our shared-memory NUMA-aware implementations of DMFI, next we compare them with a message-passing realization of the same matrix operation. Specifically, Fig. 7 reports a comparative study of the performance of the QR factorization in ScaLAPACK [7] (using version 2.2.0 of the library) and SLATE [17] (version 2022.07.00) on Rome, deploying different combinations of processes and threads per process, and launching the grid of processes using the `-map-by-socket` option of `mpirun` (OpenMPI 4.1.3 was employed as the message-passing library). Proceeding in this manner, processes are spread across NUMA nodes, and threads within processes (deployed by BLAS calls) inherit the affinity of the parent process. Following the *owner-computes* rule, this yields a NUMA-aware message-passing implementation, enabling a fair comparison with our approach. The results comprise all combinations of number of MPI processes (from 2 to 8) and process grid dimensions ($p \times q$) for ScaLAPACK. Here, instead of reporting the best performing variant in terms of number of domains for our shared-memory codes, we only report the results for the variant that matches the number of NNs for each NPS configuration.

The first observation is that, in contrast with our shared-memory codes, ScaLAPACK does not adhere to the rule of deploying as many processes as NNs are available. Contrarily, the performance for the message-passing codes keeps ramping up with the number of processes till the optimal performance point, that in all cases occurs for the highest number of processes. Especially for large matrices, our codes outperform those in ScaLAPACK for NPS1 and NPS2; however, ScaLAPACK is the best option for the optimal combination of processes and thread per process in the NPS4 case. In the case of SLATE, for brevity and given the similar qualitative behavior for the aforementioned process setups, we only report the performance of the optimal configuration. In this case, our observations yield a general inferior performance than that observed for ScaLAPACK and for our shared-memory proposal. Actually, the behavior of both ScaLAPACK and SLATE is significantly different from that of the shared-memory solution, as no performance penalty is observed in the former as the number of NNs is increased. The comparison needs to be completed by remarking that programmability is one of the key advantages of our codes compared with ScaLAPACK's (or other message-passing infrastructures), so that the lower development effort for DMFI in our case can make up for the small loss in performance.

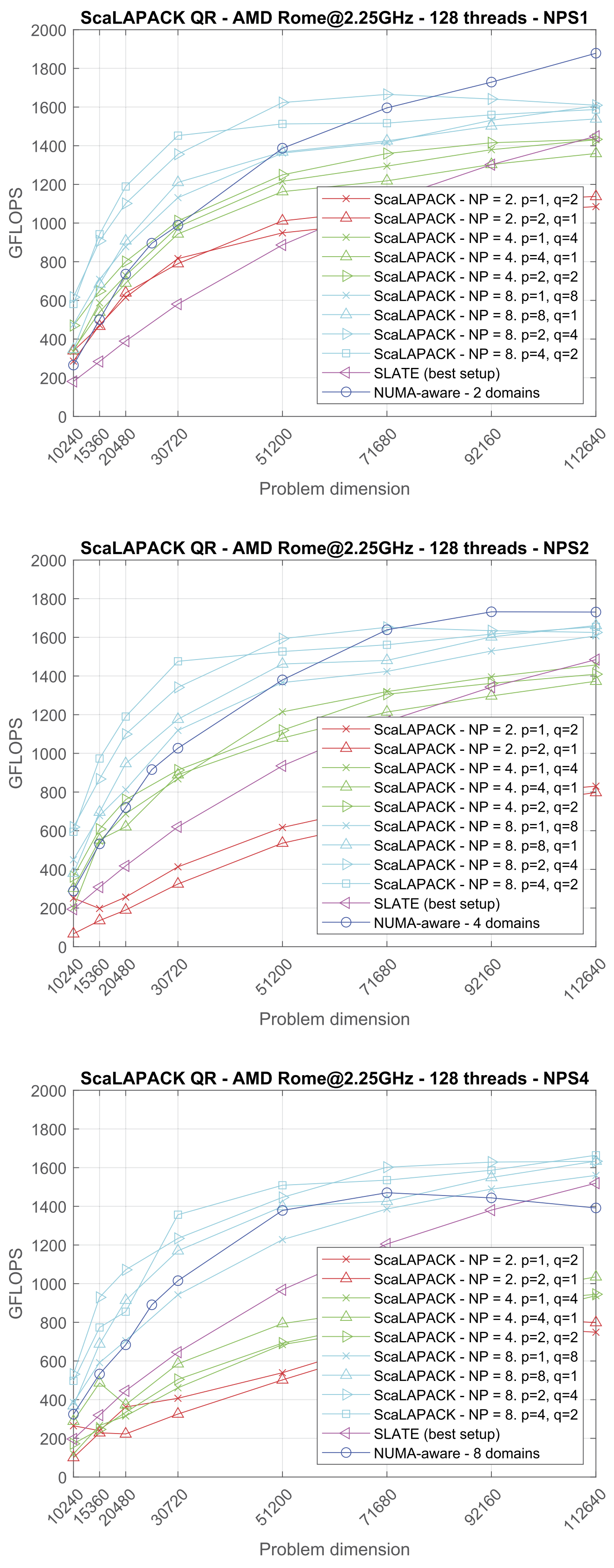


Fig. 7. Performance comparison between message-passing codes (ScaLAPACK) and our NUMA-aware shared-memory codes on Rome.

4.6. *Overall NUMA-aware DMFI performance*

Finally, we investigate on the performance of the different DMFI NUMA-aware implementations (LU, QR and Inversion) using our shared-memory approach. In this case, and for the sake of brevity, the evaluation is reported only for Kunpeng, even though similar qualitative results have been observed in Rome (actually, the performance plots reported in Figs. 4 and 5 already illustrated such a study for the case of the QR factorization in Rome).

Fig. 8 reports the performance attained by the three DMFI routines deployed in Kunpeng, featuring 4 NNs. In all cases, the experiments comprise executions for a single domain (delegating the NUMA control to the OS), and 2, 4 domains, with and without control of NUMA aspects. The qualitative results observed are similar to those observed in Rome: the peak performance is attained exclusively for the case of 4 domains with NUMA control, and this configuration, matching with the number of NNs in the system, obtains the most relevant performance boost. The performance results also reveal that the observations are consistent for the three routines, which generalizes the detailed conclusions extracted for the case of the QR factorization in Rome, and also demonstrates that our strategy is portable across different NUMA architectures (and operations).

5. Concluding remarks

In this paper, we have designed and described in detail a methodology to systematically develop parallel codes for DFMI targeting state-of-the-art NUMA architectures. Specifically, we address the complexity associated with programming novel NUMA topologies, with multiple NUMA nodes per socket and configurable memory setups on modern multi-core machines. We have extended the classic shared-memory hybrid parallelization scheme, in which parallelism is extracted simultaneously at the loop and task levels, to accommodate an execution of the algorithms that is aware of the NUMA features of the target platform. The result is a generic NUMA-aware routine for DFMI that comprises only 32 lines of code (and comments).

Our strategy is based on the formulation of multi-domain codes for DMFI, and on a proper runtime setup that fixes the computation of a specific domain to those threads that remain close (in terms of core-to-memory distance) to the data throughout the computation. Our methodology addresses the following three key aspects: *(i)* data must be logically and physically partitioned into domains that reside in different NNs; *(ii)* threads are deployed hierarchically and mapped to the corresponding NUMA node; and, *(iii)* once deployed, threads cannot migrate from one core to another. In addition, we have introduced a methodology to derive implementations which are independent of the number of domains, further easing the development process and virtually offering support any NUMA topology.

A complete performance evaluation in two modern NUMA architectures demonstrates that our NUMA-aware implementations for DMFI are valid for systems with a variable number of NNs, showing results that outperform the conventional NUMA-oblivious counterparts. In addition, we have reported performance on a par with well-known message passing libraries for dense linear algebra, with the advantage of a much simpler programming. Finally, we have demonstrated the portability of the approach showing the performance results of different DMFI.

It is known that the parallel performance of the hybrid parallelization scheme can be limited when any of the tasks is by nature difficult to parallelize at loop level and lies in the critical path of the overall operation. In the case of DMFI, this feature is usually encountered for the PF operation. Look-ahead (LA) [42]

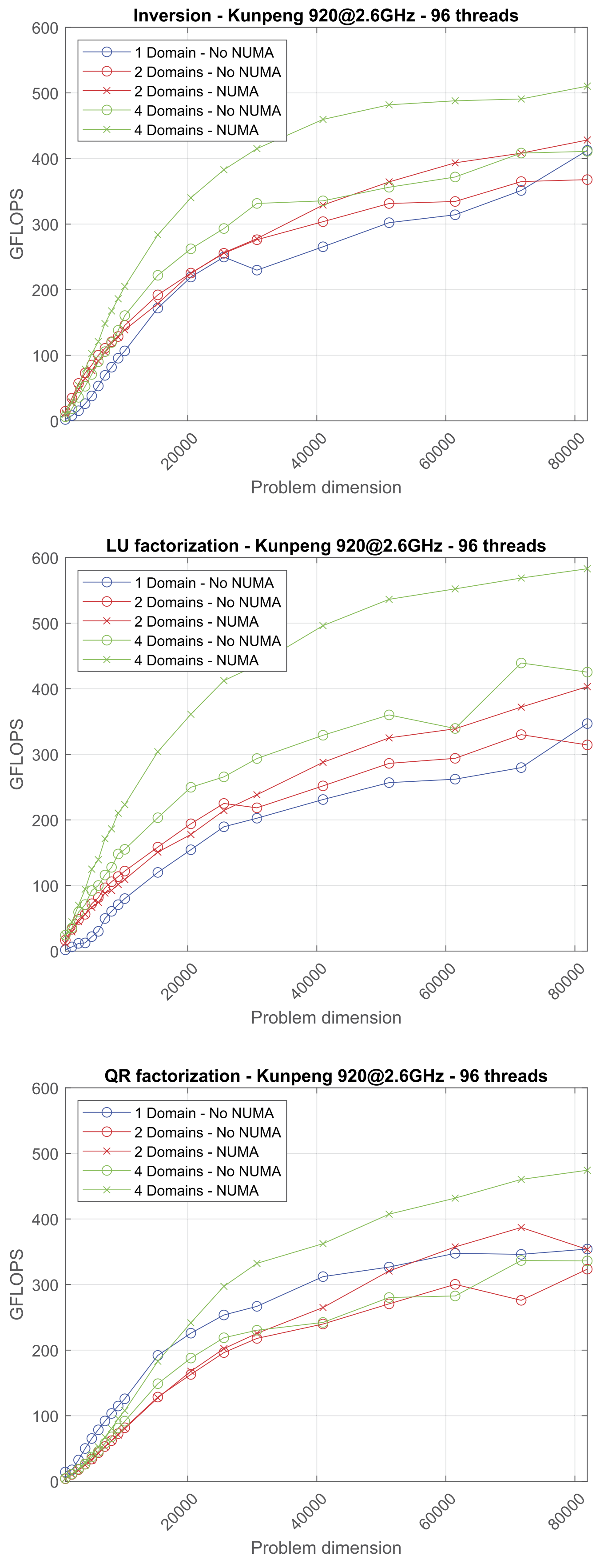


Fig. 8. Performance results for different DMFI implementations on Kunpeng.

is a technique that, integrated within DMFI, reduces the bottleneck imposed by PF by implementing a sort of software pipelining that breaks the strict dependencies between that operation and the panel updates (TU and LU) being computed in the same iteration. However, its implementation is not straightforward and presents challenges when multi-domain algorithms need to be derived. This effort is left as part of future work.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Enrique S. Quintana-Orti reports financial support was provided by European Commission. All authors reports financial support was provided by Agencia Estatal de Investigación.

Acknowledgment

This research was sponsored by project PID2019-107255GB of Ministerio de Ciencia, Innovación y Universidades; project S2018/TCS-4423 of Comunidad de Madrid; project 2017-SGR-1414 of the Generalitat de Catalunya and the Madrid Government under the Multiannual Agreement with UCM in the line Program to Stimulate Research for Young Doctors in the context of the V PRICIT, project PR65/19-22445. This project has also received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955558. The JU receives support from the European Union's Horizon 2020 research and innovation programme, and Spain, Germany, France, Italy, Poland, Switzerland, Norway. The work is also supported by grants PID2020-113656RB-C22 and PID2021-126576NB-I00 of MCIN/AEI/10.13039/501100011033 and by *ERDF A way of making Europe*.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster, cheaper, better – a hybridization methodology to develop linear algebra software for GPUs, in: W. Mei, W. Hwu (Eds.), GPU Computing Gems, vol. 2, Morgan Kaufmann, 2010, https://hal.inria.fr/inria-00547847.

[2] M.M. Ahmed, M.S. Shamim, N. Mansoor, S.A. Mamun, A. Ganguly, Increasing interposer utilization: a scalable, energy efficient and high bandwidth multicore-multichip integration solution, in: 2017 Eighth International Green and Sustainable Computing Conference (IGSC), 2017, pp. 1–6.

[3] R. Alomairy, G. Miranda, H. Ltaief, R.M. Badia, X. Martorell, J. Labarta, D.E. Keyes, Dense matrix computations on NUMA architectures with distance-aware work stealing, Supercomput. Front. Innov. 2 (1) (2015) 49–72, https://doi.org/10.14529/jsfi150103.

[4] Amd, Open-source register reference for amd family 17h processors, 2018.

[5] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, J.J. Dongarra, J.D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, D. Sorensen, LAPACK Users' Guide, third ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[6] R.M. Badia, J.R. Herrero, J. Labarta, J.M. Pérez, E.S. Quintana-Ortí, G. Quintana-Ortí, Parallelizing dense and banded linear algebra libraries using smpss, Concurr. Comput., Pract. Exp. 21 (18) (2009) 2438–2456, https://doi.org/10.1002/cpe.1463.

[7] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. Whaley, Scalapack: a portable linear algebra library for distributed memory computers - design issues and performance, in: Supercomputing '96: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, 1996, pp. 5–5.

[8] P. Caheny, M. Casas, M. Moretó, H. Gloaguen, M. Saintes, E. Ayguadé, J. Labarta, M. Valero, Reducing cache coherence traffic with hierarchical directory cache and numa-aware runtime scheduling, in: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 275–286.

[9] P. Caheny, L. Alvarez, S. Derradji, M. Valero, M. Moretó, M. Casas, Reducing cache coherence traffic with a NUMA-aware runtime approach, IEEE Trans. Parallel Distrib. Syst. 29 (5) (2018) 1174–1187, https://doi.org/10.1109/TPDS.2017.2787123.

[10] S. Catalán, F.D. Igual, R. Rodríguez-Sánchez, E.S. Quintana-Ortí, Scalable hybrid loop- and task-parallel matrix inversion for multicore processors, in: 22nd IEEE International Workshop on Parallel and Distributed Scientific and Engineering – PDSEC'21, 2021.

[11] A. Coskun, F. Eris, A. Joshi, A.B. Kahng, Y. Ma, A. Narayan, V. Srinivas, Cross-layer co-optimization of network design and chiplet placement in 2.5-d systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (12) (2020) 5183–5196, https://doi.org/10.1109/TCAD.2020.2970019.

[12] M.F. Dolz, F.D. Igual, T. Ludwig, L. Piñuel, E.S. Quintana-Ortí, Balancing task- and data-level parallelism to improve performance and energy consumption of matrix computations on the intel xeon phi, Comput. Electr. Eng. 46 (2015) 95–111, https://doi.org/10.1016/j.compeleceng.2015.06.009.

[13] S. Dominico, E.C. de Almeida, M.A.Z. Alves, J.A. Meira, Performance analysis of array database systems in non-uniform memory architecture, in: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2021, pp. 169–176.

[14] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, P. Wu, I. Yamazaki, A. Yarkhan, M. Abalenkovs, N. Bagherpour, S. Hammarling, J. Šístek, D. Stevens, M. Zounon, S.D. Relton, PLASMA: parallel linear algebra software for multicore using OpenMP, ACM Trans. Math. Softw. 45 (2) (May 2019), https://doi.org/10.1145/3264491.

[15] J.J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Softw. 16 (1) (1990) 1–17.

[16] J. Funston, M. Lorrillere, A. Fedorova, B. Lepers, D. Vengerov, J.-P. Lozi, V. Quéma, Placement of virtual containers on NUMA systems: a practical and comprehensive model, in: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18, USENIX Association, USA, 2018, pp. 281–293.

[17] M. Gates, A. Charara, J. Kurzak, A. YarKhan, M. Al Farhan, D. Sukkari, J. Dongarra, SLATE users' guide, SWAN no. 10, Tech. Rep. ICL-UT-19-01, Innovative Computing Laboratory, University of Tennessee, July 2020, revision 07-2020 https://www.icl.utk.edu/publications/swan-010.

[18] G.H. Golub, C.F. Van Loan, Matrix Computations, 3rd edition, The Johns Hopkins University Press, Baltimore, 1996.

[19] K. Goto, R.A. van de Geijn, Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw. 34 (3) (2008) 12.

[20] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, 2nd edition, Addison-Wesley, 2003.

[21] C. Imes, S. Hofmeyr, D.I.D. Kang, J.P. Walters, A case study and characterization of a many-socket, multi-tier NUMA HPC platform, in: 2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar), 2020, pp. 74–84.

[22] A. Kannan, N.E. Jerger, G.H. Loh, Enabling interposer-based disintegration of multi-core processors, in: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015, pp. 546–558.

[23] C. Lameter, NUMA (non-uniform memory access): an overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors, Queue 11 (7) (2013) 40–51, https://doi.org/10.1145/2508834.2513149.

[24] R. Laso, O.G. Lorenzo, J.C. Cabaleiro, T.F. Pena, J. Ángel Lorenzo, F.F. Rivera CIMAR NIMAR, LMMA, Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters, Future Gener. Comput. Syst. 129 (2022) 18–32, https://doi.org/10.1016/j.future.2021.11.008.

[25] X. Liu, L. Mashayekhy, Joint load-balancing and energy-aware virtual machine placement for network-on-chip systems, in: 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), 2018, pp. 124–132.

[26] G.H. Loh, N.D.E. Jerger, A. Kannan, Y. Eckert, Interconnect-memory challenges for multi-chip, silicon interposer systems, in: B.L. Jacob (Ed.), Proceedings of the 2015 International Symposium on Memory Systems, MEMSYS 2015, ACM, Washington DC, DC, USA, October 5-8, 2015, 2015, pp. 3–10, https://doi.org/10.1145/2818950.2818951.

[27] T.M. Low, F.D. Igual, T.M. Smith, E.S. Quintana-Ortí, Analytical modeling is enough for high-performance BLIS, ACM Trans. Math. Softw. 43 (2) (2016) 12.

[28] S.A. McKee, R.W. Wisniewski, Memory Wall, Springer US, Boston, MA, 2011, pp. 1110–1116.

[29] S.K. Moore, Intel's View of the Chiplet Revolution, IEEE Spectrum, April 2019.

[30] S. Naffziger, N. Beck, T. Burd, K. Lepak, G.H. Loh, M. Subramony, S. White, Pioneering chiplet technology and design for the AMD EPYC™ and Ryzen™ processor families: industrial product, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 57–70.

[31] C. Pinto, D. Syrivelis, M. Gazzetti, P. Koutsovasilis, A. Reale, K. Katrinis, H. Hofstee, ThymesisFlow: a software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE Computer Society, Los Alamitos, CA, USA, 2020, pp. 868–880.

[32] M. Plauth, F. Eberhardt, A. Grapentin, A. Polze, Improving the accessibility of NUMA-aware C++ application development based on the PGASUS framework, Concurr. Comput., Pract. Exp. e6887 (2022), https://doi.org/10.1002/cpe.6887.

[33] M. Popov, A. Jimborean, D. Black-Schaffer, Efficient thread/page/parallelism autotuning for NUMA systems, in: Proceedings of the ACM International Conference on Supercomputing, ICS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 342–353.

[34] G. Quintana-Ortí, E.S. Quintana-Ortí, R.A. van de Geijn, F.G. Van Zee, E. Chan, Programming matrix algorithms-by-blocks for thread-level parallelism, ACM Trans. Math. Softw. 36 (3) (2009) 14, https://doi.org/10.1145/1527286.1527288, http://doi.acm.org/10.1145/1527286.1527288.

[35] B.M. Rogers, A. Krishna, G.B. Bell, K. Vu, X. Jiang, Y. Solihin, Scaling the bandwidth wall: challenges in and avenues for CMP scaling, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, Association for Computing Machinery, New York, NY, USA, 2009, pp. 371–382.

[36] P. Roy, S.L. Song, S. Krishnamoorthy, A. Vishnu, D. Sengupta, X. Liu, NUMA-caffe: NUMA-aware deep learning neural networks, ACM Trans. Archit. Code Optim. 15 (2) (jun 2018), https://doi.org/10.1145/3199605.

[37] I. Sánchez Barrera, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, Graph partitioning applied to dag scheduling to reduce numa effects, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 419–420.

[38] J. Schwarzrock, H.M.G. de, A. Rocha, A.C.S. Beck, A.F. Lorenzon, Effective exploration of thread throttling and thread/page mapping on NUMA systems, in: 2020 IEEE 22nd International Conference on High Performance Computing and Communications, IEEE 18th International Conference on Smart City, IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2020, pp. 239–246.

[39] Y.S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S.G. Tell, Y. Zhang, W.J. Dally, J. Emer, C.T. Gray, B. Khailany, S.W. Keckler, Simba: scaling deep-learning inference with multi-chip-module-based architecture, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '52, Association for Computing Machinery, New York, NY, USA, 2019, pp. 14–27.

[40] T.M. Smith, et al., Anatomy of high-performance many-threaded matrix multiplication, in: Proc. IEEE 28th Int. Parallel and Distributed Processing Symp., IPDPS'14, 2014, pp. 1049–1059.

[41] Socket SP3 Platform NUMA Topology for AMD Family 17h Models 30h–3Fh, https://developer.amd.com/wp-content/resources/56338_1.00_pub.pdf, 2019.

[42] P. Strazdins, A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization, Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

[43] X. Su, F. Lei, Hybrid-grained dynamic load balanced GEMM on NUMA architectures, Electronics 7 (12) (2018), https://doi.org/10.3390/electronics7120359.

[44] Universal Chiplet Interconnect Express (UCIe), https://www.uciexpress.org. (Accessed March 2022).

[45] F.G. Van Zee, R.A. van de Geijn, BLIS: a framework for rapidly instantiating BLAS functionality, ACM Trans. Math. Softw. 41 (3) (2015) 14.

[46] G. Voron, Efficient Virtualization of NUMA Architectures. (Virtualisation Efficace d'Architectures NUMA), Ph.D. thesis, Sorbonne University, Paris, France, 2018.

[47] W.A. Wulf, S.A. McKee, Hitting the memory wall: implications of the obvious, SIGARCH Comput. Archit. News 23 (1) (1995) 20–24, https://doi.org/10.1145/216585.216588.

[48] J. Xia, C. Cheng, X. Zhou, Y. Hu, P. Chun, Kunpeng 920: the first 7-nm chiplet-based 64-core ARM SoC for cloud services, IEEE MICRO 41 (5) (2021) 67–75, https://doi.org/10.1109/MM.2021.3085578.

[49] W. Zhang, Z. Jiang, Z. Chen, N. Xiao, Y. Ou, NUMA-aware DGEMM based on 64-bit ARMv8 multicore processors architecture, Electronics 10 (16) (2021), https://doi.org/10.3390/electronics10161984.

[50] J. Zhao, N.E. Jerger, M. Ga, What can chiplets bring to multi-tenant clouds?, in: Cloud@MICRO Workshop in Conjunction with MICRO, 2021.

Sandra Catalán received the B.Sc. degree, M.Sc. degree in Intelligent Systems and Ph.D. in Computer Science in 2012, 2013 and 2018, respectively, from the Universitat Jaume I of Castell, Spain. In 2018, she moved as a postdoctoral researcher to Barcelona Supercomputing Center, and in 2019 she joined the Universidad Complutense de Madrid where she is currently Assistant Professor. Her current research is focused on energy saving on moderate-scale clusters and low-power processors, parallel algorithms for numerical linear algebra and asymmetric architectures.

Francisco D. Igual obtained a bachelor degree in Computer Engineering from University Jaume I de Castelln (Spain) in 2006, and a Ph.D. degree in Computer Science from the same University in 2011. In 2011, he moved as a postdoctoral researcher to The University of Texas at Austin, and in 2012 he joined the Universidad Complutense de Madrid where he is currently Associate Professor. His research interests include high-performance and energy-aware computing, dense linear algebra library development and optimization, and runtime task scheduling on massively heterogeneous architectures.

José R. Herrero received bachelor and Ph.D. degrees in Computer Science from Universitat Politècnica de Catalunya (UPV), Spain, in 1993 and 2006, respectively. He currently holds a position as Associate Professor in the Computer Architecture Department at UPC. His current research interests include high performance scientific computing, advanced architectures and accelerators, parallel programming and mixed precision computing.

Rafael Rodríguez-Sánchez obtained his M.S. and Ph.D. degrees in Computer Science from the University of Castilla-La Mancha, Spain, in 2010 and 2013, respectively. He is currently Assistant Professor at Universidad Complutense de Madrid. His research interests include video coding, parallel programming, heterogeneous computing, and task parallelism.

Enrique S. Quintana-Ortí received bachelor and Ph.D. degrees in Computer Science from Universitat Politècnica de València (UPV), Spain, in 1992 and 1996, respectively. After more than 20 years at Universitat Jaume I, he is currently Professor in Computer Architecture at UPV. His current research interests include parallel programming, linear algebra, energy consumption, transprecision computing and deep learning as well as advanced architectures and hardware accelerators.