Parallel assembly of finite element matrices on multicore computers

Petr Krysl*[1]

[a] *University of California, San Diego, Department of Structural Engineering, 9500 Gilman Dr #0085, La Jolla, 92093, CA, USA*

Abstract

A complete suite of algorithms to implement parallel assembly of sparse finite element matrices on multicore computers is presented. The approach is broken down into six algorithms. The goal is to cater both to nonlinear computations, where it is possible to amortize sequential algorithms over multiple assembly passes, and linear computations, where sequential algorithms cannot be amortized. Therefore, we consider the entire process, starting with the mesh, and finishing with the sparse matrix in the compressed column format, in order to compare meaningfully parallel and sequential implementations. Importantly, the entire construction of the sparse matrix is parallelized. Numerical experiments are carried out on five distinct multicore computers (including A64FX and Nvidia Grace systems) with up to 144 cores, and the results indicate that decent scaling can be obtained up to 48 computing threads. The algorithms are described in the programming language Julia, and their complete source code is available on Github.

Keywords:

PK wishes to dedicate this work to his doctoral advisor, Prof. Zdeněk Bittnar, of the Czech Technical University in Prague, on the occasion of his 80th birthday.

1. Introduction

Not everyone has access to a supercomputer with thousands of processing units, or to a powerful workstation with a bank of GPGPUs (ie. general

Email address: pkrysl@ucsd.edu (Petr Krysl*)

purpose graphics processing units), but practically everyone has a computer with multiple processing units (cores) at their fingertips. To exploit such resources, modern analysis codes therefore necessarily need to identify operations that can be executed in parallel, and then find suitable algorithms and implementations to realize speedups.

It is by now understood that in order to achieve scalability to large numbers of processing units (hundreds or thousands), be it CPUs or GPGPUs, sequential finite element software needs to be rebuilt from scratch. The substantial associated cost can be estimated from the size of the teams involved in such efforts [1, 2]. Further, not everyone will be able or willing to throw away their software and know-how and start from zero. Techniques for exploiting parallelism in *existing* software with *little effort* are valuable. The present work is aimed at shared memory environments with multiple cores available to perform the construction of the requisite sparse matrices in parallel. Readers interested in massively parallel finite element computations are kindly directed to [3].

In this paper the focus is on acceleration of finite element assembly of sparse matrices. For very-high-order methods the preferred approach is to work with matrix free methods [1, 4]. On the other hand, lower-order methods often rely on the availability of sparse matrices that represent the required operators. Thus, the ability to construct such matrices in a scalable manner is desirable, because little can be gained by employing a parallel equation solver, when a substantial sequential calculation is still present.

The solver usually dictates the format of the sparse matrix [5, 6, 7, 8]. We may mention the

- Coordinate format (COO): Simple and flexible scheme, where data is stored in three arrays – values, row indices, and column indices of the non-zero entries.

- Compressed Sparse Row (CSR): Three arrays are used: column indices of non-zero elements and the associated values, and pointers to where each row starts in the index array.

- Compressed Sparse Column (CSC): As in CSR, but interchanging rows and columns.

- Block Sparse Row (BSR): Intended for sparse matrices where the non-zero elements are grouped in blocks.

- Diagonals: Mostly used for structured matrices.
- ELLPACK: Used for banded matrices, with many variants.
- Skyline: Older format developed for direct solvers.

But the list of more or less esoteric options is much longer [6, 9].

For unstructured meshes, the COO, CSR, CSC, and the BSR formats appear to dominate. There are many tradeoffs concerning the performance for various applications. The tradeoff of most interest here pertains to the expense of constructing the data structures of the matrix: starting from scratch, the COO is the least troublesome to construct (however, it requires more memory than for instance CSC and CSR, and a subsequent conversion into a format suitable for the solution phase is required). Further, the expense of assembling elementwise quantities to the global matrix is crucial. The costs here arise from the need to avoid conflicts (race conditions) due to concurrent write access. The access control can be addressed in various ways: critical section, atomic updates, block locking, and sequencing of assembly based on mesh coloring [10, 11, 12, 13, 14, 15]. The COO format is also amenable to conflict-free assembly [16, 17].

The assembly of finite element sparse matrices has been studied quite extensively, including recently (viz just listed references and [18, 11, 19, 20, 21, 22]). A common feature of these works is the assumption that matrix structure is available ab initio, or has been constructed in a sequential manner before the parallel assembly starts. As an example, we may refer to the WorkStream as a design pattern suitable for shared-memory environments with multiple cores [21] (identified for use in the well-known deal.II package [2]). The best alternative pattern therein adopted mesh coloring, and good speedups could be achieved. The improved performance was mostly attributed to the conflict-free access to the global sparse matrix, enabled by the embarrassingly parallel nature of the processing of totally independent degrees of freedom associated with elements of the same color. Nevertheless, the initial matrix structure was generated in the PETSC library in a serial computation.

While the papers mentioned above deal with multithreaded assembly, only an isolated pair of papers address *all* processes necessary for the parallel construction of a sparse matrix: Dziekonski et al. [16, 17] assemble entries of the sparse matrix into the COO format, which can be done in a parallel process with no synchronization, at the cost of a partitioning of the mesh into separate sets. The COO matrix is then converted to the CSR format using a bespoke threaded version of an UMFPACK routine (the algorithm is sketched out in [17]).

The goal of the present paper is to describe parallel algorithms applied not only during the assembly phase, but also to the construction of the sparsity pattern. We take a different route than Dziekonski and coworkers: a CSC matrix structure is constructed in parallel, and the assembly is then based on element coloring to avoid write conflicts. One advantage relative to [16, 17] is that the need to store the large COO arrays is eliminated.

We are able to parallelize a decisive amount of serial computation (with the exception of operations such as allocation of memory and such). Consequently, the complete construction of the sparse matrix scales reasonably well with the number of threads. The plan for this presentation is to explain in Section 2 the algorithms using actual code snippets. The computations were coded in the Julia programming language [23]. The complete implementation is open source. In Section 3 we present numerical experiments, and investigate the speedups and reasons for observed decrease of parallel efficiency for larger number of employed threads. Section 4 offers some points for discussion. We conclude by proposing extensions.

2. Methods

The computations described below were implemented in the Julia programming language [24, 23], in the framework of the `FinEtools.jl` Julia package [25]. Since the Julia programming language is eminently suited to high-performance numerical computation, and hence is of general interest to the readers by itself, we shall briefly explain some features of the codes.

The `FinEtools.jl` package furnishes basic support for definition and manipulation of meshes, element types and access to basis functions and their derivatives, computation of Jacobian matrices and Jacobians, numerical integration, definition of fields on meshes, etc. Its algorithms are purely serial (sequential). The generic support of finite element methods in `FinEtools.jl` is employed, in what we call “application packages”, to acoustics, linear vibration, heat conduction, statics and dynamics of shells, vibration in fluids, etc. [26].

The intended target of the parallelization described here is to enable any of the application packages to construct the requisite sparse matrices (acoustic mass and stiffness, mass and elastic stiffness, conductivity and capacity, and so on) by using a serial code *without change*, yet to achieve *speedups on multicore machines.*

Our assumptions can be summarized as follows:

- Nodal methods are considered (the degrees of freedom reside at the nodes). All nodes carry the same number of degrees of freedom.
- Galerkin methods are the focus: hence the sparse matrix will always be symmetric.
- The compressed sparse column matrix (CSC) format is considered. But, for symmetric matrices the compressed sparse row (CSR) matrix format can be obtained for free from the compressed sparse column data.
- The integration of the weighted-residual equations is performed element by element. This leaves out for instance nodal integration methods [27, 28]. Research would be needed as to a suitable partitioning of the domain, since the nodal operators need to be calculated on the surrounding patch of elements.

The application code calls functions such as that shown in Figure 1 to construct a sparse matrix. Here `femm` is the so called finite element model machine (FEMM). The compiler of Julia dispatches on the concrete type of the FEMM in order to execute code specialized to a particular formulation: for instance, mean-strain versus incompatible modes versus plain isoparametric etc. The FEMM also provides access to the information that supports numerical integration such as the definition of the connectivity of the mesh, the numerical integration rule, and the code in general-purpose FEMMs is dimension-independent and quantities such as Jacobian matrices can be calculated for any manifold dimension of the finite elements using ostensibly an

```
K = stiffness(femm, assmblr, geom, u)
```

Figure 1: The call to execute sparse matrix assembly in a serial fashion.

identical line of code (in reality the calculation is dispatched to an appropriate implementation based on the types of the arguments)[1].

The geometric information is accessed as a field defined on the finite element mesh, `geom`, and the displacement field `u` is passed in to allow access to the degree of freedom information. Importantly, the sparse-matrix assembler `assmblr` again has a type which is used for dispatch. Various types of assemblers are needed in general: symmetric and unsymmetric matrices, general and diagonal matrices, forms of lumped matrices in the case of mass matrices, and so on.

The application packages use assemblers that work with three vectors, `I`, `J`, and `V`, that represent a sparse matrix in the COO format. This representation is then converted to a default format in the Julia `SparseArrays` package, the compressed sparse column (CSC), by a call to the `sparse` function, which takes the COO vectors as arguments. All of these operations are serial, and for large matrices this computation can become a bottleneck. Even when this computation is not a very expensive operation, as soon as everything else is executed in parallel, even single operation done entirely in sequential fashion can thoroughly destroy parallel scaling [29].

The design of the application serial code shown in Figure 1 provides an opening to adapt these calls to assemble the sparse matrix in parallel. Firstly, the finite element model machines can work on subdomains of the overall finite element model. The assembly then can proceed in parallel over all the subdomains, provided race conditions to update entries of the global sparse matrix concurrently by multiple processes can be avoided. Secondly, we can create an assembler capable of constructing directly the compressed sparse column matrix, again in parallel.

It will be useful to clearly state our intention: We wish to present algorithms that start from the same data (the mesh) and end up with the same

[1] The so called "multiple dispatch" is a very powerful feature of Julia. As the authors of [23] state:

> This paradigm is a natural fit for numerical computing, since so many important operations involve interactions among multiple values or entities. Binary arithmetic operators are obvious examples, but many other uses abound. The fact that the compiler can pick the sharpest matching definition of a function based on its input types helps achieve higher performance, by keeping the code execution paths tight and minimal.

sparse global matrix both in the serial (sequential) and in the parallel version. This is not always the case in the literature: Oftentimes authors make an assumption that the sparse matrix data structure has been constructed before the parallel assembly begins, and therefore take into account only scaling that has to do with that parallel phase [21, 22, 13]. That may be a perfectly valid stance when the serial computation phase can be amortized over many assembly passes, such as in nonlinear computations, where the Jacobian needs to be repeatedly recomputed. Here we take a different position: we are interested in accelerating linear computations also. Therefore, to investigate scaling on multiple cores, any and all phases of the computation, serial or parallel, must be taken into account. Thus our intent is to fix the start and end point of the serial and parallel computation to be the same, so that thus obtained speedups reflect the true costs of the parallel implementation relative to the sequential one.

2.1. Overall parallel algorithm

The parallel sparse matrix assembly can be summarized in actual Julia code as shown in Figure 2.

```julia
function parallel_make_matrix(
    fes, dofnums, ndofs, FT, n2e, # data
    createsubd, matrixupdt!,      # functions
    ntasks                        # how many threads?
)
    n2n = FENodeToNeighborsMap(n2e, fes)        # ALG 1
    matrix = csc_symmetric_pattern(dofnums,     # ALG 2
                                 ndofs, n2n, FT)
    e2e = FElemToNeighborsMap(n2e, fes)         # ALG 3
    colors = element_coloring(fes, e2e, ntasks) # ALG 4
    decomposition = decompose(fes, colors,      # ALG 5
                              createsubd, ntasks)
    return parallel_matrix_assembly!(           # ALG 6
        SysmatAssemblerSparsePatt(matrix),
        decomposition,
        matrixupdt!,
    )
end
```

Figure 2: The overall algorithm of parallel sparse matrix assembly. In Julia, the character # introduces a comment.

The lines 6, 7, 9, 10, 11, and 13 in Figure 2 correspond to distinct algorithms that will be described in the following sections. In brief, a map from nodes to their neighbors (through their connections by finite elements) in line 6 facilitates the construction of the sparse matrix pattern on line 7. In order to avoid race conditions when assembling the computed values to the matrix, the finite elements are separated into distinct colors, so that no node is shared by more than one element of a given color (line 10). The coloring is speeded up with a data structure of lists of elements connected to a given element through the nodes (line 9). The elements are then separated into distinct colors, and within each color into equal-size sets that can be accessed concurrently (line 11). Finally, the subdomains are processed by running a function to calculate and assemble entries of the stiffness matrix on line 13, color by color, and concurrently for all elements of the same color.

The following arguments are supplied in Figure 2:

1. `fes`: Finite element set, enabling access to the connectivity of the mesh, giving for each finite element the numbers of the connected nodes. Furthermore, the finite element set encodes the basis functions, manifold dimension, methods for computing the Jacobian matrix, etc.;
2. `dofnums`: Array that describes the distribution of the degrees of freedom in a nodal field. There is one row for each node, and as many columns as there are degrees of freedom per node;
3. `ndofs`: Total number of degrees of freedom, that is the row and column dimension of the sparse matrix;
4. `FT`: Type of the entries of the sparse matrix (typically `Float64`)[2];
5. `n2e`: Map from finite element nodes to the elements that share it. A vector of vectors of element numbers;
6. `createsubd`: Function that creates a "finite element machine" from a set of finite elements;
7. `matrixupdt!`: Function that computes the elementwise matrices and assembles them to the global sparse matrix object;
8. `ntasks`: Number of tasks to employ in the decomposition of the elements all the same color into subsets to be processed concurrently; typically equal to the number of threads that have been spun up to run the simulation.

Next we proceed to describe the individual algorithms.

[2] In Julia types may be passed around as data.

2.2. ALG 1: Build node to neighbors map

The algorithm is invoked on line 6 of Figure 2. The map is simply a vector of vectors: the map stores in the location k a vector of numbers of nodes that are connected to the given node k (so in a way it is just another form of the CSR matrix). The line 6 in Figure 2 is executed by using the function in Figure 3: note that it relies on the availability of the incidence relation $(0, 3)$ (nodes-to-elements, referred to as `n2e`), which is the transpose of the usual mesh connectivity (incidence relation) of the type $(3, 0)$ [30]. As indicated, `n2e` is a vector of vectors, listing for each node the numbers of the elements that are connected to it. As shown in Figure 3, the node to neighbors map will list for each node the nodes that are connected to it by finite elements. This information can be collected from the $(0, 3)$ `n2e` incidence relation in parallel: note the `@threads` macro[3] that schedules the execution of the loop over `i` concurrently on as many threads as are available. In Figure 3, `conn` is a vector of tuples of node numbers. [4]

```
function FENodeToNeighborsMap(
    n2e::N2EMAP,
    fes::FE,
) where {N2EMAP<:FENodeToFEMap,FE<:AbstractFESet}
    return FENodeToNeighborsMap(_make_map(n2e, fes.conn))
end

function _make_map(n2e, conn)
    npe = length(conn[1])
    map = Vector{eltype(n2e.map)}(undef, length(n2e.map))
    Threads.@threads for i in eachindex(n2e.map)
        map[i] = _nneighbors(i, n2e.map[i], conn, npe)
    end
    return map
end
```

Figure 3: ALG 1: The parallel construction of the node to neighbor map.

The list of neighbors for node `i` is constructed as shown in Figure 4: For each node we simply loop over all the elements connected to the node, and

[3] Macros run at compile time, producing code to be compiled. In this case, the “for” loop is transformed to run on multiple threads.

[4] The `where` keyword is used to define a parametric type.

collect all the nodes that define the connectivity `conn` of the elements into a vector (Figure 4). In this figure, `npe` is the number of nodes per element, a fixed quantity as assumed here. Finally, the temporary vector with duplicates is sorted and pruned to contain only unique values.

```
function _nneighbors(self, ellist, conn, npe)
    totn = length(ellist) * (npe - 1) # w/o self-ref
    nodes = fill(zero(eltype(conn[1])), totn)
    p = 1
    @inbounds for i in ellist
        for k in conn[i]
            if k != self # w/o self-reference
                nodes[p] = k; p += 1
            end
        end
    end
    return unique!(sort!(nodes))
end
```

Figure 4: The function to compile the list of unique node numbers that are the neighbors of a node shared by all the elements in the list `ellist`. That list will not include a self-reference.

2.3. ALG 2: Create the sparsity pattern

This algorithm, invoked on line 7 of Figure 2, creates a sparse matrix represented in the CSC format (Figure 5). We could refer to this matrix as the sparsity pattern, because while it does represent all entries of the matrix that could potentially be nonzero, in reality it is a zero sparse matrix.

The function `_csc_arrays` on line 7 of Figure 5 creates the three arrays that constitute the data structure of the CSC matrix. Its logic is described in Figure 6. Here `IT` and `FT` are the types of the elements of the arrays that define CSC matrix, integer and floating point numbers respectively.

The function `_row_blocks` (shown in Figure 7) sweeps through the columns in parallel, employing all available threads, using the node to neighbors map, and the knowledge of how many degrees of freedom there are per node, `nd`, to compute the lengths of the blocks of row indexes. Here `j` is the global degree of freedom number for node `k` and for the local degree of freedom number `d` (such as coordinate direction in 3D elasticity, where `d` is 1, 2, 3). Note that the assumption is that all degrees of freedom at one node interact with all degrees of freedom at all its neighbors.

```
function csc_symmetric_pattern(
    dofnums::Array{IT,2},
    nrowscols,
    n2n,
    FT = Float64,
) where {IT<:Integer}
    colptr, rowval, nzval =
        _csc_arrays(IT, FT, n2n.map, dofnums)
    Base.Threads.@threads for n in axes(dofnums, 1)
        _store_dofs!(rowval, n, n2n.map[n],
                     dofnums, colptr)
    end
    return SparseMatrixCSC(nrowscols, nrowscols,
                           colptr, rowval, nzval)
end
```

Figure 5: ALG 2: The sequence of operations to create the symmetric CSC sparsity pattern.

```
1 function _csc_arrays(IT, FT, map, dofnums)
2     colptr = _row_blocks(IT, map, dofnums)
3     psp_scan!(colptr) # cum sum
4     sumlen = colptr[end] - 1
5     rowval = Vector{IT}(undef, sumlen) # initialize later
6     nzval = _zeros_via_calloc(FT, sumlen)
7     return colptr, rowval, nzval
8 end
```

Figure 6: Logic of the function to prepare the three arrays, `colptr`, `rowval`, `nzval`, that describe the sparse CSC pattern.

Next, on line 3 of Figure 6, follows a parallel "scan" (aka cumulative sum) `psp_scan!`, to compute the entries of the `colptr` array. It is implemented as a multithreaded segmented scan in `FinEtoolsMultithreading.jl` [31], and it is executed on all available threads. The array `colptr` is thus initialized, and the `rowval` can be created, uninitialized, since it will be built in the next step, while the array `nzval` can be created lazily initialized to zero, using the `calloc` system function.

The algorithm in Figure 5 continues with a parallel loop over each node[5]

[5] `axes(dofnums , 1)` is the range of all the rows of the array `dofnums`. That is all the

```
function _row_blocks(IT, map, dofnums)
    nd = size(dofnums, 2)
    total_dofs = length(map) * nd
    lengths = Vector{IT}(undef, total_dofs + 1)
    lengths[1] = 1
    @inbounds Threads.@threads for k in eachindex(map)
        kl = (length(map[k]) + 1) * nd # +1 for self
        for d in axes(dofnums, 2)
            j = dofnums[k, d]
            lengths[j+1] = kl
        end
    end
    return lengths
end
```

Figure 7: The calculation of the row-index block lengths for each column.

to populate blocks of row numbers using the degrees of freedom (lines 9–12 of Figure 5). For each node, the function `_store_dofs!` fills in the blocks of row index numbers (array `rowval`) for all degrees of freedom (columns) at that node (refer to Figure 8). The row indexes are first collected in the loop on line 4 of Figure 8 (note that the self-reference of the node `n` is included, and the concatenation of `n` with the vector `nghbrs` is carried out with `LazyArrays` [32] without any allocation of memory), and then they are sorted in-place (line 10), and copied to blocks corresponding to all the other degrees of freedom at the node (line 11).[6]

The function to create the sparsity pattern (Figure 5) completes with the (sequential) constructor of the CSC object[7] on line 13. At this point, the sparsity pattern (i.e. a matrix which is set up to store the non-zero values, but which is at this point a representation of a zero matrix) is ready for the adding of the values.

nodes `n = 1, 2, ...size(dofnums, 1)`.

[6] The `@view` macro eliminates copying the slice of the array. The `@inbounds` macro suppresses bounds checking within a loop. `sort!` sorts the array in place.

[7] A composite type in Julia is a collection of named fields, an instance of which can be treated as a single value. Calling it an object does not mean the same thing as in, for instance, C++: In Julia, all values are objects, but functions are not bundled with the objects they operate on.

```julia
function _store_dofs!(rowval, n, nghbrs, dofnums, colptr)
    s1 = colptr[dofnums[n, 1]]
    p = 0
    @inbounds for k in ApplyArray(vcat, n, nghbrs)
        for d in axes(dofnums, 2)
            rowval[s1+p] = dofnums[k, d]; p += 1
        end
    end
    bl = p # block length
    sort!(@view(rowval[s1:s1+bl-1]))
    @inbounds for d in 2:size(dofnums, 2)
        s = colptr[dofnums[n, d]]
        copy!(@view(rowval[s:s+bl-1]),
              @view(rowval[s1:s1+bl-1]))
    end
    return nothing
end
```

Figure 8: Function to initialize the indexes of the rows for all columns at node `n`.

2.4. ALG 3: Build element-to-neighbors map

The element-to-neighbors map defines the graph for the computation of the element coloring. In this incidence relation, the list of adjacent elements is constructed for each element in the mesh, where elements are adjacent if they share at least one node. Similarly to the node-to-neighbors map, this data structure can also be constructed in parallel, without any need for synchronization. In the interest of brevity, the code to compute the data structure is omitted, but the complete algorithm can be found in Reference [31].

2.5. ALG 4: Element coloring

Graph coloring is an assignment of colors to the vertices of a graph such that no two adjacent vertices get the same color. The vertices here are the finite elements, and the edges are the shared nodes between elements. Several parallel implementations exist, mostly based on the seminal Jones-Plassmann approach [33]. Here we adopt an improvement of the original algorithm based on shortcutting, which increases the available parallelism [34]. The original C++ graph-coloring library is accessible in Julia through a thin wrapper [35]. The input graph is constructed from the representation of the graph as the element-to-neighbors data structure computed above.

```
function _make_femms(singlecolorfes, ntasks, createsubd)
    chks = chunks(1:count(singlecolorfes), ntasks)
    return [createsubd(subset(singlecolorfes, ch))
                                 for (ch, j) in chks]
end
```

Figure 10: Function to create finite element machines from finite elements that are of a given color. The function `chunks` computes the partitioning of the range into pieces of roughly the same size [36].

subset of the finite elements `fesubset`. An instance of the material is also supplied. Julia returns the result of the last expression, no need for a return statement.

```
function createsubd(fessubset)
    FEMMDeforLinearMSH8(
        DeforModelRed3D,
        IntegDomain(fessubset, GaussRule(3, 2)),
        material
    )
end
```

Figure 11: Function to create a finite element machine from a subset of the finite elements that comprise the overall mesh.

2.7. ALG 6: Assembly of values

As shown in Figure 12, the logic of the function on line 13 of Figure 2 is a simple loop, enclosed in a synchronization block. For each level in the decomposition (i.e. for each color in the element coloring), the computation over the decomposition of the elements of the same color is run in parallel by spawning a separate thread, executing the function `matrixupdt!`. This function mutates the assembler `assmblr` by adding the nonzero values to the sparse matrix (Figure 13).

The assembler in Figure 2, `SysmatAssemblerSparsePatt`, is another component that needed to be developed for the parallel assembly: the assemblers in `FinEtools` all work with a COO matrix, whereas this one needs to add numbers to the CSC zero-matrix pattern. The complete implementation can be found in Reference [31]. The elementwise matrices are computed, and

```
function parallel_matrix_assembly!(
    assmblr,
    decomposition,
    matrixupdt!::F,
) where {F<:Function}
    for femms in decomposition
        Threads.@sync begin
            for femm in femms
                Threads.@spawn matrixupdt!(femm, assmblr)
            end
        end
    end
    return assmblr._pattern
end
```

Figure 12: ALG 6: The parallel loop to add values to the sparse matrix.

then their elements (entries) are immediately stored into the `nzval` array, using a binary search within the row numbers array, `rowval` (both arrays are constituents of the CSC matrix). Note that the threads run concurrently without any contention since they work with elements of the same color. [8]

The `matrixupdt!` function computes the contribution to the sparse matrix for each subdomain (Figure 13).

```
function matrixupdt!(femm, assmblr)
    associategeometry!(femm, geom)
    stiffness(femm, assmblr, geom, u)
end
```

Figure 13: The function to assemble a sparse matrix contribution from a subdomain of the mesh (in this case, one partition of a subset of elements of the same color).

2.8. Theoretical parallel scaling

The overall parallel algorithm in Figure 2 is in its effect entirely equivalent to a sequential construction of the sparse matrix. Therefore, provided all the

[8] Two additional macros to execute tasks in parallel are introduced in Figure 12: `Threads.@sync` for synchronization and `Threads.@spawn` for spawning tasks on available threads.

steps scale perfectly with the number of threads, namely the time taken for N threads is the original time for 1 thread divided by N, we should expect the perfect speedup of N for the parallel construction of a sparse matrix.

That happens very rarely in parallel computing. In fact, sequential operations are clearly still present in the algorithms shown above, for instance allocations of arrays, but these operations typically tend to be fairly inexpensive. Moreover, limitations of the hardware, and some optimal software implementation, will to some degree degrade the performance (for instance by the inability to supply data to the CPUs at a sufficient rate). This is dictated by Amdahls's formula [29]. We explore the scaling in the next section experimentally.

3. Experiments

The test case is a linear static analysis of a stubby corbel modelled with mean-strain hexahedral 8-node finite elements with energy-sampling stabilization [37]. As such, the elements are integrated with a single-point rule (basic response) and at the same time with a $2 \times 2 \times 2$ Gauss's integration rule (stabilization). The meshes utilized for the test cases are summarized in Table 1.

The computations described below were implemented in the Julia programming language [24, 23], in the framework of the `FinEtools.jl` Julia package [25]. The implementation of the algorithms of the parallel sparse matrix assembly may be found in the package `FinEtoolsMultithreading.jl` [31].

The physical experiments have been carried out on several machines. The first three represent machines of common provenance, while the last two are somewhat exotic, found nearly exclusively at high performance computing institutions:

H Linux machine with 64 AMD Opteron(tm) Processor 6380 cores at 1.4 GHz, with 256 GB of DDR3 RAM, and caches L1d: 1 MiB (64 instances), L1i: 2 MiB (32 instances), L2: 64 MiB (32 instances), L3: 48 MiB (8 instances).

F Linux machine with two sockets of Intel(R) Xeon(R) CPU E5-2670 @ 2.60GHz (8 cores each, or 32 virtual cores total), 256GB of DDR4 memory, and caches L1d: 512 KiB (16 instances), L1i: 512 KiB (16 instances), L2: 4 MiB (16 instances), L3: 40 MiB (2 instances).

S Apple M2 Ultra, 16 performance cores (+ 8 efficiency cores), 192 GB of unified memory.

Test case	Number of elements	Number of dofs	Number of non zeros
50	$500,000$	1.54M	$120,912,904$
100	$4,000,000$	12.14M	$969,655,804$

Table 1: Test case parameters.

- **A** Fujitsu A64FX, 48 cores in four NUMA nodes, 32 GB of high-bandwidth memory (HBM2 with 1 TB/s bandwidth). This was one node on the supercomputer Ookami.
- **G** Dual socket NVIDIA Grace superchip (144 cores, 72 per CPU), L1: 64KB I-cache + 64KB D-cache per core, L2: 1MB per core, L3: 234MB per superchip. 960 GB of LPDDR5X ECC memory.

Computer systems A and G were accessed at the Institute for Advanced Computational Science at Stony Brook University. Systems H, F, and S could be used in single-user mode (private machines). The runs are labelled for instance as (H-50), meaning the simulation with mesh 50 was run on machine H.

Julia uses pthreads for multithreading. The graph-colouring library uses OpenMP threads internally. No threads were allocated to the BLAS library. The computational threads were not pinned to hardware resources in any of the runs. The number of threads was chosen up to the maximum number of threads available.

3.1. Results on a single thread

First we will demonstrate that even when the parallel matrix assembly is run on a single thread, it is not hopelessly inefficient. Table 2 lists timings for six test cases (three machines times two meshes) when the assembly is executed sequentially. In that case, the sparse matrix is first assembled in the COO format, and then converted using a library function to the CSC format.

In Table 3 we now show the timings (the total and its constituents) for the presented parallel assembly technique (i.e. the program outlined in Figure 2) when it is run on each machine using only a single computing thread. Even when a single computing thread is used, we still compute the element coloring and the supporting data structures and the domain decomposition. Not surprisingly, then, the total time is greater than the total time for the sequential assembly. However, the difference is relatively slight (at most 20%).

Test case	Total assembly	Assemble COO	Convert to CSC
(H-50)	55.4	42.8	12.5
(H-100)	459.1	338.3	120.4
(F-50)	30.5	21.4	9.0
(F-100)	250.5	176.9	73.6
(S-50)	8.8	7.1	1.6
(S-100)	70.8	56.3	14.5

Table 2: Serial assembly timing

Test case	Total assembly	Neighbor maps	Sparsity	Colors	Add values
(H-50)	58.4	5.6	1.4	0.2	51.1
(H-100)	467.7	47.9	11.3	1.27	403.8
(F-50)	33.8	3.4	0.8	0.1	29.4
(F-100)	274.6	29.4	6.6	0.7	235.7
(S-50)	10.4	1.0	0.2	0.04	9.1
(S-100)	88.1	9.3	1.7	0.3	76.7

Table 3: Parallel assembly timing carried out by a single task.

3.2. Results on multiple threads

It is well known that just 10% of the overall computing cost being serial will limit the parallel speedup on 32 processors to less than 8 (Amdahl's formula) [29]! Let us call T_s the total time needed for the serial workload, and T_p the total time needed for the parallelizable workload on a single computing thread. Then the speedup using N computing threads (i.e. independent processing units) can be estimated as

$$S(N) = \frac{T_s + T_p}{T_s + T_p/N} = \frac{T_s/T_p + 1}{NT_s/T_p + 1} N \tag{1}$$

If $T_s > 0$, the speedup will be less than ideal, $S(N) < N$. The factor

$$E(N) = S(N)/N = \frac{T_s/T_p + 1}{NT_s/T_p + 1} \tag{2}$$

is the so called parallel efficiency [38, 29].

In order to handle jitter in the results due to random events within the environment of the test machines, each test was run 5 times, and the minimum timing was extracted from the records. In all figures the **execution**

time of the stiffness matrix assembly constituent parts and the total are reported in seconds in the left sub-figure. **Parallel efficiency** of the stiffness matrix assembly constituent parts and of the total are reported in the right sub-figure.

In the interest of saving space, the **legend** to Figures 14–22 is included here: —⋎— node to neighbor map, —+— element to neighbor map, —✱— domain decomposition, —○— element coloring, —▲— construct sparse zero, —◆— add values to matrix, —□— total assembly.

We begin by inspecting results obtained with the Apple M2 Ultra (machine S). In Figure 14 on the left we can see the timing results for machine S and test mesh 100. The assembly time is clearly dominated by the assembly of values (line 13 of the listing in Figure 2). The next items on the graph are the construction of the neighbor lists, and these execute an order of magnitude faster than the assembly of values. The construction of the sparse matrix pattern scales well up to 8 threads. The decomposition is the cheapest operation, and also stops to scale for > 9 threads. The element coloring scales less than optimally, but it is an inexpensive operation.

The parallel efficiency (on the right of the figure) is reported for each of the constituent parts of the algorithm. We do not get perfect speedup, neither for the individual constituents, nor for the total assembly time: for 16 threads, the actual overall speedup is less than $0.8 \times 16 \approx 13$. However, we can see that the assembly of the values into the sparsity pattern scales fairly well, which bodes well for applications to nonlinear simulations: In that case the element coloring, the construction of the sparsity pattern, the domain decomposition, and the construction of the supporting data structures would be all amortized over repeated assembly passes (the only constituent left would be the addition of values).

Figure 15 presents analogous information obtained with a much smaller mesh on the same machine (test case S-50). We can see that the observations made for the much larger computation with the test case S-100 hold without change.

Figure 16 shows the timing and parallel efficiency for the large mesh test case (100) executed on the machine F. The timing curve of the overall assembly time is reasonably straight up to 15 threads. A similar story is repeated in Figure 17 for the smaller mesh (50) on machine F.

Machine H had the slowest cores, but a relatively large number of them (up to 64). For the larger mesh, 100, the overall assembly time decreases with reasonable parallel efficiency up to 32 threads. Then the parallel effi-

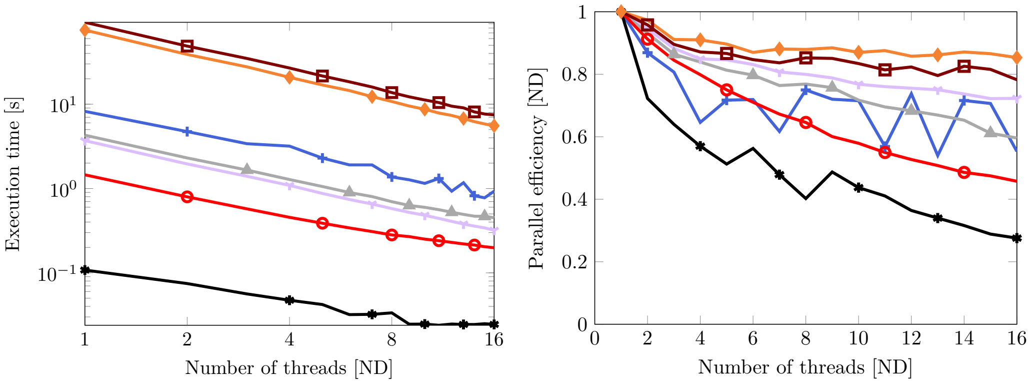

Figure 14: Test case (S-100). Executed with 1, ..., 16 threads. Refer to the legend in the text.

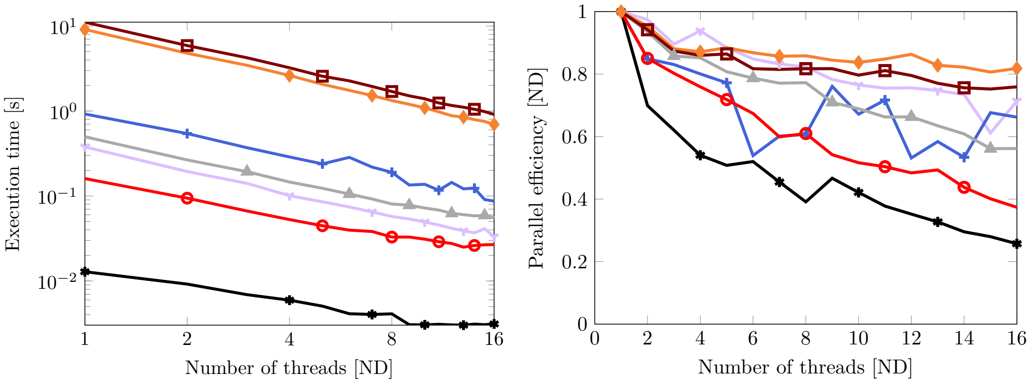

Figure 15: Test case (S-50). Executed with 1, ..., 16 threads. Refer to the legend in the text.

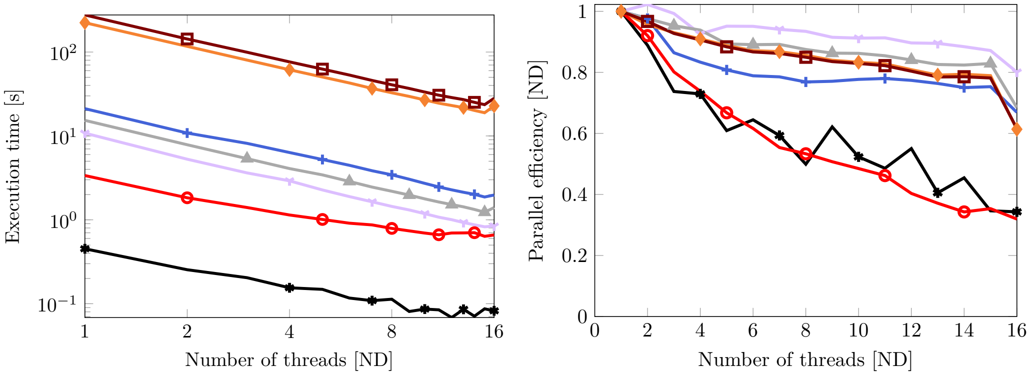

Figure 16: Test case (F-100). Executed with 1, ..., 16 threads. Refer to the legend in the text.

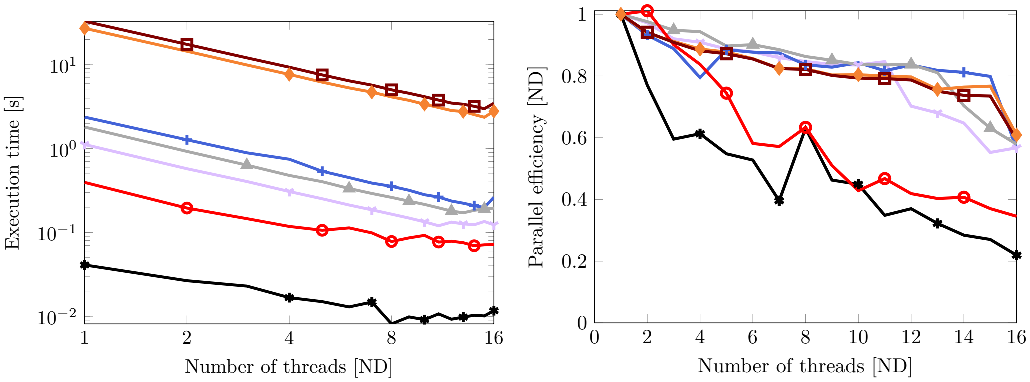

Figure 17: Test case (F-50). Executed with 1, ..., 16 threads. Refer to the legend in the text.

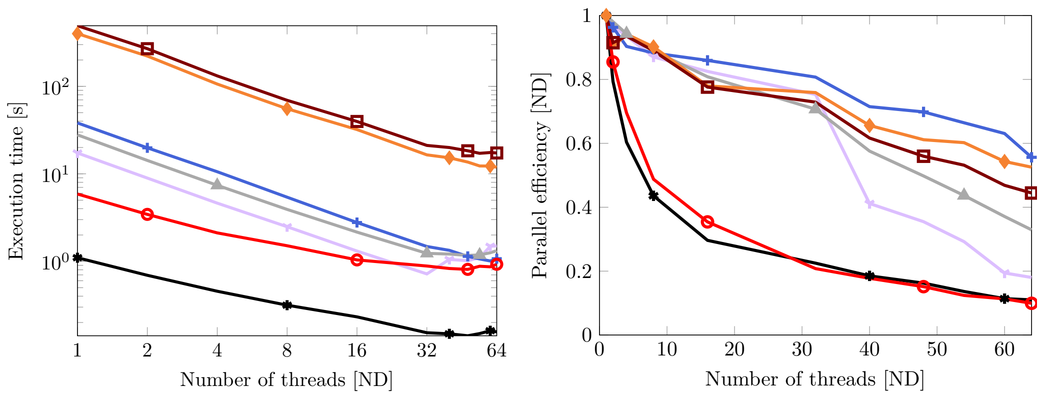

Figure 18: Test case (H-100). Executed with 1, 2, 4, 8, 16, 32, 40, 48, 54, 60, and 64 threads. Refer to the legend in the text.

ciency drops to < 0.5 for 64 threads. Progressively, the construction of the neighbor data structures stops scaling when more then 32 threads are used. A possible reason may be the irregular memory access: at some point the data cannot be moved to caches sufficiently quickly from the main memory. Similar observations can be made in Figure 19 for the smaller mesh, except that the deterioration of the parallel efficiency is somewhat quicker.

Machine A (Fujitsu A64FX) had a limited amount of memory, which did not allow for the larger-mesh (100) simulation to run. Nevertheless, on the smaller mesh the achieved scaling was excellent, all the way up to 48 threads (overall efficiency of > 0.8). The outstanding performance is likely due to the high bandwidth memory (HBM2), as also noted in [39]. It should be noted that while this processor is indeed excellent at number crunching, it is not so good at other tasks, such as compilation [40].

Finally, the Grace dual-socket superchip is a relative newcomer. The Grace CPU Superchip had 144 Arm Neoverse V2 cores in two sockets. According to the tuning guide, the bottlenecks when moving data between the chips should have been removed with the NVLink Chip-2-Chip (C2C) interconnect. Furthermore, the processor features the NVIDIA Scalable Coherency Fabric (SCF), which is a mesh fabric and distributed cache architecture to scale cores and bandwidth. To maintain data flow between the CPU cores, the NVLink-C2C, and the memory, the SCF reportedly provides over 3.2 TB/s of total bisection bandwidth. The Grace CPU Superchip has only

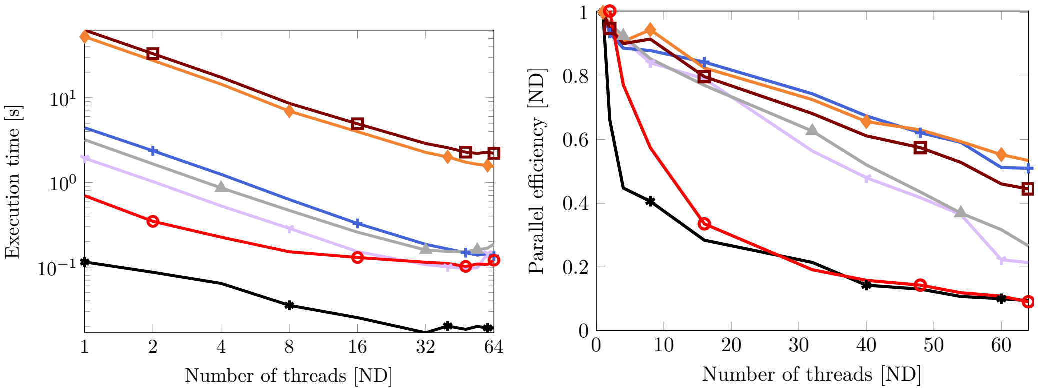

Figure 19: Test case (H-50). Executed with 1, 2, 4, 8, 16, 32, 40, 48, 54, 60, and 64 threads. Refer to the legend in the text.

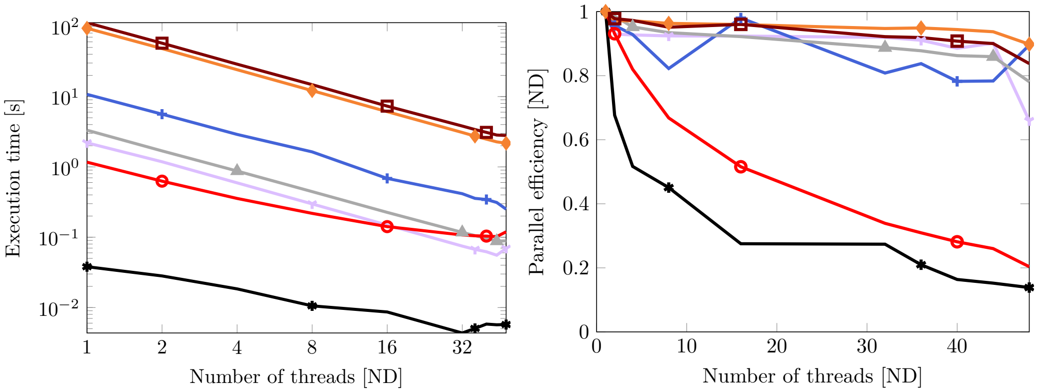

Figure 20: Test case (A-50). Executed with 1, 2, 4, 8, 16, 32, 36, 40, 44, and 48 threads. Refer to the legend in the text.

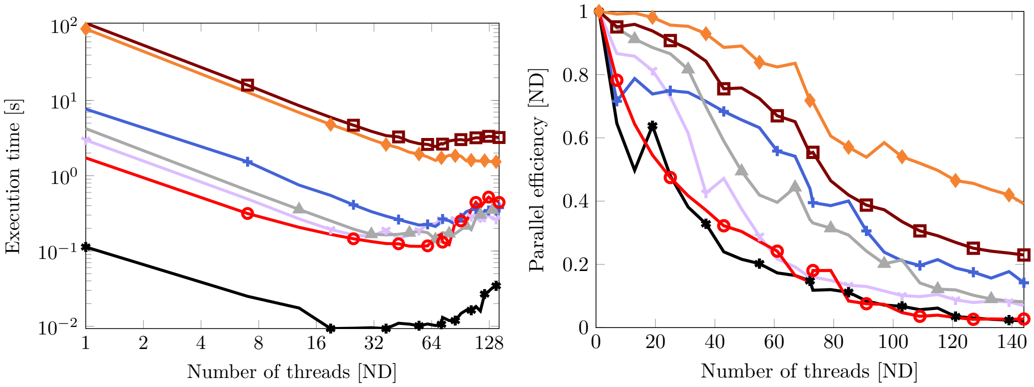

Figure 21: Test case (G-100). Executed with from 1 by 6 to 139 threads, and also for 72, 144 threads. Refer to the legend in the text.

two NUMA nodes [41].

From this description, it is natural to have high expectations of such a system. Figure 21 therefore disappoints. All constituent parts of the overall algorithm drop off fairly quickly in parallel efficiency above 32 threads, and the parallel efficiency drops to ≈ 0.6 for 64 threads. When using more than 72 threads, associating those threads with hardware resources means that some of these threads will need to reside on the second socket (which is the second NUMA domain!). So up to 72 threads, the computation can reside on a single socket. The parallel efficiency deteriorates towards 72 threads, but for more than that the overall assembly time no longer decreases. This behavior is amplified in Figure 22 for the test case with the smaller mesh.

The reasons for these observations are unclear. Additional investigations showed the same lack of scaling across the CPUs for other softwares run in the shared-memory mode on threads [42]. It will be interesting to decipher how the lack of performance should be attributed to the characteristics of the hardware, but the details are somewhat unclear since these systems are just coming online.

An implementation that would scale well on the Grace dual-socket would then presumably employ mesh partitioning and MPI to distribute the computation to the two sockets, running multiple threads per socket. However, we are then no longer talking about "simple" multithreaded execution.

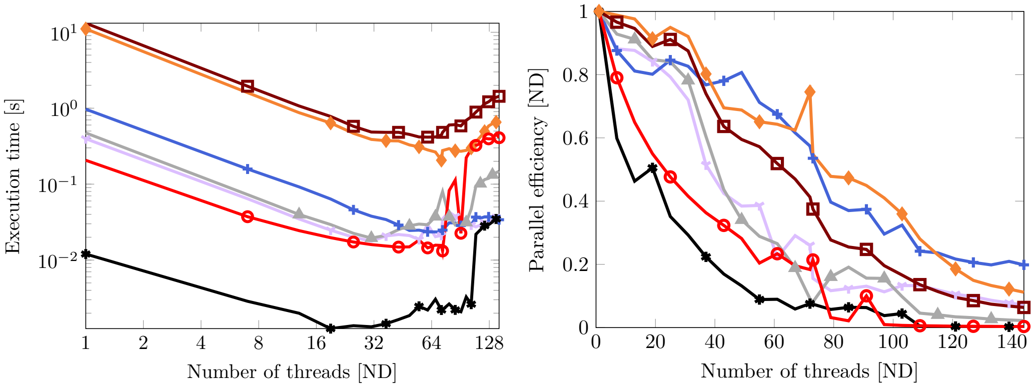

Figure 22: Test case (G-50). Executed with from 1 by 6 to 139 threads, and also for 72, 144 threads. Refer to the legend in the text.

4. Discussion

Table 4 summarizes the speedups that are available for a sparse matrix assembly in linear problems, where the assembly is done once, and in nonlinear problems, where the assembly is repeated many times. In the nonlinear case, the only constituent is the assembly of the values into the sparsity pattern. Hence, the speedups are slightly better in the nonlinear case. The machine A demonstrates that it is possible to get remarkable speedups, provided the hardware and software are up to the task.

Why does the parallel efficiency decrease despite the theoretical guarantees of good scaling? Scaling is probably controlled by the available memory bandwidth, and that is likely to decrease with an increased number of threads. In general, an algorithm like matrix assembly that is computation bound in sequential execution will then get increasingly memory-bandwidth bound during a parallel execution. That is likely what we see in the figures presented in the previous section.

5. Conclusions

A complete suite of algorithms to implement parallel assembly of sparse finite element matrices on multicore computers was presented. The approach was described as a succession of six algorithms: construction of neighborhood data structures, (1) node to neighbors and (2) element to neighbors,

Test case	Linear (one-off) assembly	Nonlinear assembly (repeated)
H-50	28.5	34.2
H-100	28.5	33.6
F-50	9.5	9.7
F-100	9.8	9.8
S-50	12.1	13.1
S-100	12.5	13.7
A-50	40.2	43.1
G-50	29.7	53.7
G-100	39.7	51.7

Table 4: Parallel assembly speedups. 64 threads were used for machine H, 16 computing threads were used for machines F and S, and 48 threads were used for machine A. Machine G was employed with 72 threads (to stay on a single socket).

(3) computation of the sparsity pattern, (4) element coloring, (5) domain decomposition, and (6) computation and assembly of values into the sparse matrix.

All algorithms were parallelized. Numerical experiments were carried out on five distinct multicore computers with up to 144 cores. Good scaling could be obtained up to 48 computing threads, but reasonable speedups were observed even for 64 threads. The challenge now is to extend the scalability further, given that hundreds of computing cores may be soon commonplace. It is very likely that this will not be possible without rethinking some of the algorithms, taking into account the flow of data across the memory hierarchies.

Since the parallel algorithm and the sequential algorithms start from the same data (the mesh), and construct the same data structure (sparse matrix), a complete picture of possible speedups in linear problems could thus be drawn. The conclusion was that indeed it was possible to apply the developed parallel algorithms in linear problems profitably.

The algorithms were described in the programming language Julia, and in order to support reproducibility, the complete source code was made available on Github.

5.1. Acknowledgements

This work used Ookami at Stony Brook University through allocation MTH240014 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program [43], which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. The authors would like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a National Science Foundation grant (#1927880).

References

[1] A. Abdelfattah, V. Barra, N. Beams, R. Bleile, J. Brown, J.-S. Camier, R. Carson, N. Chalmers, V. Dobrev, Y. Dudouit, P. Fischer, A. Karakus, S. Kerkemeier, T. Kolev, Y.-H. Lan, E. Merzari, M. Min, M. Phillips, T. Rathnayake, R. Rieben, T. Stitt, A. Tomboulides, S. Tomov, V. Tomov, A. Vargas, T. Warburton, K. Weiss, GPU algorithms for efficient exascale discretizations, Parallel Computing 108 (2021) 102841. doi:https://doi.org/10.1016/j.parco.2021.102841.

[2] D. Arndt, W. Bangerth, M. Bergbauer, M. Feder, M. Fehling, J. Heinz, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, B. Turcksin, D. Wells, S. Zampini, The deal.II library, version 9.5, Journal of Numerical Mathematics 31 (3) (2023) 231–246. doi:doi:10.1515/jnma-2023-0089.

[3] W. Bangerth, C. Burstedde, T. Heister, M. Kronbichler, Algorithms and data structures for massively parallel generic adaptive finite element codes, ACM Trans. Math. Softw. 38 (2) (jan 2012). doi:10.1145/2049673.2049678.

[4] J. Martínez-Frutos, P. J. Martínez-Castejón, D. Herrero-Pérez, Fine-grained GPU implementation of assembly-free iterative solver for finite element problems, Computers & Structures 157 (2015) 9–18. doi:https://doi.org/10.1016/j.compstruc.2015.05.010.

[5] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. van der Vorst, Templates for the Solution of Algebraic Eigenvalue Problems, Society for Industrial and Applied Mathematics, 2000. arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898719581, doi:10.1137/1.9780898719581.

[6] D. Langr, P. Tvrdík, Evaluation criteria for sparse matrix storage formats, IEEE Transactions on Parallel and Distributed Systems 27 (2) (2016) 428–440. doi:10.1109/TPDS.2015.2401575.

[7] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. R. Bishop, A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units, SIAM Journal on Scientific Computing 36 (5) (2014) C401–C423. doi:10.1137/130930352.

[8] G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, S. J. Sherwin, Finite element assembly strategies on multi-core and many-core architectures, International Journal for Numerical Methods in Fluids 71 (1) (2013) 80–97. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.3648, doi:https://doi.org/10.1002/fld.3648.

[9] J. Wong, E. Kuhl, E. Darve, A new sparse matrix vector multiplication graphics processing unit algorithm designed for finite element problems, International Journal for Numerical Methods in Engineering 102 (12) (2015) 1784–1814. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.4865, doi:https://doi.org/10.1002/nme.4865.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.4865

[10] L. Thébault, E. Petit, Q. Dinh, Scalable and efficient implementation of 3d unstructured meshes computation: a case study on matrix assembly, SIGPLAN Not. 50 (8) (2015) 120–129. doi:10.1145/2858788.2688517.

[11] U. Kiran, D. Sharma, S. S. Gautam, GPU-warp based finite element matrices generation and assembly using coloring method, Journal of Computational Design and Engineering 6 (4) (2018) 705–718. arXiv:https://academic.oup.com/jcde/article-pdf/6/4/705/33135627/j.jcde.2018.11.001.pdf, doi:10.1016/j.jcde.2018.11.001.
URL https://doi.org/10.1016/j.jcde.2018.11.001

[12] Z. Fu, T. James Lewis, R. M. Kirby, R. T. Whitaker, Architecting the finite element method pipeline for the GPU, Journal

of Computational and Applied Mathematics 257 (2014) 195–211. doi:https://doi.org/10.1016/j.cam.2013.09.001.
URL https://www.sciencedirect.com/science/article/pii/S0377042713004470

[13] M. Bošanský, B. Patzák, Parallelization of assembly operation in finite element method, Acta Polytechnica 60 (1) (2020) 25–37. doi:10.14311/AP.2020.60.0025.

[14] J. D. Trotter, X. Cai, S. W. Funke, On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core CPUs, ACM Trans. Math. Softw. 48 (2) (may 2022). doi:10.1145/3503925.

[15] A. Sky, C. Polindara, I. Muench, C. Birk, A flexible sparse matrix data format and parallel algorithms for the assembly of finite element matrices on shared memory systems, Parallel Computing 117 (2023) 103039. doi:https://doi.org/10.1016/j.parco.2023.103039.

[16] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski, Finite element matrix generation on a GPU, Progress In Electromagnetics Research 128 (2012) 249–265. doi:10.2528/PIER12040301.

[17] A. Dziekonski, P. Sypek, A. Lamecki, M. Mrozowski, Generation of large finite-element matrices on multiple graphics processors, International Journal for Numerical Methods in Engineering 94 (2) (2013) 204–220. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.4452, doi:https://doi.org/10.1002/nme.4452.

[18] C. Cecka, A. J. Lew, E. Darve, Assembly of finite element methods on graphics processors, International Journal for Numerical Methods in Engineering 85 (5) (2011) 640–669. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2989, doi:https://doi.org/10.1002/nme.2989.

[19] I. Gribanov, R. Taylor, R. Sarracino, Parallel implementation of implicit finite element model with cohesive zones and collision response using CUDA, International Journal for Numerical Methods in Engineering 115 (7) (2018) 771–790. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.5825, doi:https://doi.org/10.1002/nme.5825.

URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.5825

[20] K. Banaś, F. Krużel, J. Bielański, Finite element numerical integration for first order approximations on multi- and many-core architectures, Computer Methods in Applied Mechanics and Engineering 305 (2016) 827–848. doi:https://doi.org/10.1016/j.cma.2016.03.038.
URL https://www.sciencedirect.com/science/article/pii/S0045782516301256

[21] B. Turcksin, M. Kronbichler, W. Bangerth, WorkStream – a design pattern for multicore-enabled finite element computations, ACM Trans. Math. Softw. 43 (1) (aug 2016). doi:10.1145/2851488.

[22] I. Z. Reguly, M. B. Giles, Finite element algorithms and data structures on graphical processing units, Int. J. Parallel Program. 43 (2) (2015) 203–239. doi:10.1007/s10766-013-0301-6.

[23] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, SIAM review 59 (1) (2017) 65–98.

[24] The Julia Project, The julia programming language, https://julialang.org/ (Accessed 03/13/2024).

[25] Petr Krysl, FinEtools: Finite Element tools in Julia, https://github.com/PetrKryslUCSD/FinEtools.jl (Accessed 03/13/2024).

[26] Petr Krysl, FinEtoolsOrg: Finite Element Repositories, https://github.com/FinEtoolsOrg/FinEtoolsRepos/blob/master/README.md (Accessed 05/13/2024).

[27] P. Krysl, B. Zhu, Locking-free continuum displacement finite elements with nodal integration, International Journal for Numerical Methods in Engineering 76 (7) (2008) 1020–1043. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.2354, doi:https://doi.org/10.1002/nme.2354.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2354

[28] R. Sivapuram, P. Krysl, On the energy-sampling stabilization of Nodally Integrated Continuum Elements for dynamic analyses, Finite Elements in Analysis and Design 167 (2019) 103322. doi:https://doi.org/10.1016/j.finel.2019.103322.
URL https://www.sciencedirect.com/science/article/pii/S0168874X19303154

[29] R. Robey, Y. Zamora, Parallel and High Performance Computing, Manning, 2021.

[30] P. Krysl, Lightweight finite element mesh database in Julia, Advances in Engineering Software 157-158 (2021) 103005. doi:https://doi.org/10.1016/j.advengsoft.2021.103005.

[31] Petr Krysl, FinEtoolsMultithreading: Multithreading for Finite Element tools in Julia, https://github.com/PetrKryslUCSD/FinEtoolsMultithreading.jl (Accessed 03/13/2024).

[32] JuliaArrays, LazyArrays: Lazy arrays and linear algebra in Julia, https://github.com/JuliaArrays/LazyArrays.jl (Accessed 05/13/2024).

[33] M. T. Jones, P. E. Plassmann, A parallel graph coloring heuristic, SIAM Journal on Scientific Computing 14 (3) (1993) 654–669. doi:10.1137/0914041.

[34] G. Alabandi, E. Powers, M. Burtscher, Increasing the parallelism of graph coloring via shortcutting, in: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 262–275. doi:10.1145/3332466.3374519.
URL https://doi.org/10.1145/3332466.3374519

[35] Petr Krysl, ECLGraphColor: Parallel graph coloring in Julia, https://github.com/PetrKryslUCSD/ECLGraphColor.jl (Accessed 05/09/2024).

[36] JuliaFolds2, ChunkSplitters: Simple chunk splitters for parallel loop executions, https://github.com/JuliaFolds2/ChunkSplitters.jl (Accessed 05/09/2024).

[37] P. Krysl, Mean-strain 8-node hexahedron with optimized energy-sampling stabilization, Finite Elements in Analysis and Design 108 (2016) 41–53. doi:https://doi.org/10.1016/j.finel.2015.09.008.

[38] D. Eager, J. Zahorjan, E. Lazowska, Speedup versus efficiency in parallel systems, IEEE Transactions on Computers 38 (3) (1989) 408–423. doi:10.1109/12.21127.

[39] M. A. S. Bari, B. Chapman, A. Curtis, R. J. Harrison, E. Siegmann, N. A. Simakov, M. D. Jones, A64FX performance: experience on Ookami, in: 2021 IEEE International Conference on Cluster Computing (CLUSTER), 2021, pp. 711–718. doi:10.1109/Cluster48925.2021.00106.

[40] M. Giordano, M. Klöwer, V. Churavy, Productivity meets performance: Julia on A64FX, in: 2022 IEEE International Conference on Cluster Computing (CLUSTER), 2022, pp. 549–555. doi:10.1109/CLUSTER51413.2022.00072.

[41] Anonymous, NVIDIA Grace performance tuning guide, Tech. Rep. DA-11438-001_04, NVIDIA (March 2024).

[42] M. Giordano, Personal communication, Advanced Research Computing Centre at the University College London (2024).

[43] T. J. Boerner, S. Deems, T. R. Furlani, S. L. Knuth, J. Towns, ACCESS: Advancing innovation: NSF's advanced cyberinfrastructure coordination ecosystem: Services & support, in: Practice and Experience in Advanced Research Computing, PEARC '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 173–176. doi:10.1145/3569951.3597559.
URL https://doi.org/10.1145/3569951.3597559