Optimization and distributed-memory parallelization of the finite element library Femlisp

von

M. Heisig & N. Neuss

No. 405 2019
Optimization and Distributed-Memory Parallelization of the Finite Element Library Femlisp

Marco Heisig*, Nicolas Neuss †

April 15, 2019

Femlisp is a versatile finite element toolkit for solving partial differential equations. We describe the transition from a purely single-threaded application to one with both multithreaded parallelism and distributed parallelism and present a performance analysis for a 3D elasticity problem. Femlisp is written in Common Lisp, but the parallelization techniques we employ are relevant for any scientific software project that is written in a dynamic language.

1 Introduction

Finite Elements (FE) are a particularly successful method for solving partial differential equations (PDEs), which, in turn can be used to model many applications in our continuum world, cf. [12]. But, as partial differential equations are inherently complex, also the Finite Element Method (FEM) inherits this complexity. This becomes strikingly obvious when one looks for FE variants that emphasize efficiency or even optimality in several dimensions. For example, the most economic approximations in terms of the number of unknowns require adaptivity in the mesh (h-adaptivity) as well as the polynomial order (p-adaptivity). This enforces using unstructured meshes and matrices without a regular structure, which has the drawback of preventing many standard optimization and parallelization techniques.

Another problem is the software design of FE libraries considering that they are attacking PDE problems which are

*Informatik 10, FAU Erlangen-Nürnberg, Cauerstraße 11, D-91058 Erlangen, Tel.: +49-9131-85-67288, Fax: +49-9131-85-28928
†Angewandte Mathematik 3, FAU Erlangen-Nürnberg, Cauerstraße 11, D-91058 Erlangen, Tel.: +49-9131-85-67208, Fax: +49-9131-85-67201
1. not wholly understood mathematically
2. where each one can appear in a large variety of subclasses
3. and where even the aims and goals are often not clearly specified.

In this situation, it is obvious that a general FE library must accommodate to different requirements during its lifetime, which requires using a design and an implementation which is as flexible and dynamic as possible. However, this is apparently in conflict with low-level optimization and parallelization.

Now, the fundamental motivation behind the FE library Femlisp \[14, 21\] which radically chose Lisp (or more precisely, Common Lisp) as implementation language, because it is one of the very few computer languages which give the user an interactive use with high-level code together with the possibility to write also low-level efficient code in the same language. Furthermore, the possibility of syntactical code transformations link both aspects by providing conversions from high-level flexible code into low-level efficient code at compile-time. This is very similar to the features of the much more recently introduced language Julia \[8\], while most other approaches use a mix of computer languages for achieving both goals (cf. \[13, 15\]).

While Femlisp was a serial application for a long time, we announced a parallelization effort in \[17\]. This effort has now reached a state where we can solve nontrivial PDE problems with a performance comparable with other PDE solvers. In this contribution, we want to show this for a model problem which features some special difficulties, but also several advantages which can be exploited to achieve a rather satisfactory compromise between high-level flexibility and low-level performance.

2 Benchmark problem

2.1 Model problem

Many materials in modern technology are in fact composites, sometimes with regular, sometimes with irregular or even random structure. Such materials exhibit large-scale behavior which is different from that of each of the components, and which depends both on the properties of the components, but also on their geometric arrangement.

In this situation, it is an important task to derive macroscopic behavior of such materials from their microscopic composition, and the already well-established mathematical field concerned with such derivations is called homogenization (see, e.g., the monographs \[7\], \[1\]). In the following, we consider such a homogenization problem for the case of an elastic composite with periodic structure. The main reason for choosing this problem is that we considered a very similar problem already ten years ago in \[22\], but at that time we could not compute the 3D case with the high accuracy we wanted to achieve.
More precisely, we consider a domain consisting of periodic cubic cells where each cell consists of a matrix enclosing a hole. From this geometry and the elastic properties of the matrix, one can calculate the effective properties which a material would exhibit, that consists of a large number of small and regularly arranged cells.

Phenomenologically, this calculation consists of averaging the responses of a representative cell to several imposed stress situations. Mathematically, it consists of solving the following system of partial differential equations: Find periodic functions $N_{qr} : Y \rightarrow \mathbb{R}$ with $1 \leq q,l,r \leq 3$ which satisfy

$$- \frac{\partial}{\partial y_i} (A_{ij}^{kl}(y) (\delta_{jq} \delta_{lr} + \frac{\partial N_{qr}^l(y)}{\partial y_j})) = 0, \quad y \in Y,$$

on the domain $Y$ which is a 3-dimensional unit cube from which a ball of diameter $\frac{1}{2}$ is cut out in the middle (the 2D analogue is shown in Fig 1). $A_{ij}^{kl}$ are the $3^4$ components of the elasticity tensor. In the model case considered here, they are chosen such that they describe an isotropic medium with Lamé parameters $\lambda = 1$ and $\mu = 1$. The solution of this system is defined only up to 27 additive constants (which do not change the result of formula (3) below). For having a completely well-defined system, we fix these constants by setting

$$N_{qr}^l(0,0,0) = 0, \quad l, q, r \in \{1, 2, 3\}.$$  

Then the macroscopic/effective elasticity tensor can be obtained by simply averaging certain quantities (the microscopic stresses) involving those functions:

$$\hat{A}_{ij}^{kr} = \int_Y A_{ij}^{kl}(y) (\delta_{jq} \delta_{lr} + \frac{\partial N_{qr}^l(y)}{\partial y_j}) \, dy. \quad (3)$$

### 2.2 Solution algorithm

The continuous problem (1), (2) has certain qualities which must be taken into account for treating it effectively. First, the underlying domain $Y$ is not as trivial as a cube, but it has the nice property that it is smoothly bounded (the corners and edges of the cube do not count as such, because
of the periodic boundary conditions). This implies that all components of the tensor $N_{lr}^q$ are arbitrarily smooth, and can be approximated well if one uses discretizations which can adapt to the smooth boundary. Second, the tensor $N_{lr}^q$ consists of 27 component functions ($3 \cdot 3 = 9$ stress situations yielding each a 3D-deformation), which gives some leverage for exploiting the vectorization capabilities of temporary computer architectures.¹

A good choice of discretization which is fit for approximating the smooth boundary and the smooth tensor are Lagrange finite elements of high order $p$ which also approximate the domain using exact cell maps. These are implemented in the current FEMLISP version 2.0.1 [14]) which was released at the end of March 2015, and where such an approximation of the effective elasticity tensor is available even as an interactive FEMLISP demo. A fixed order $p = 5$ is used, and the resulting discretized system is solved by the conjugate gradient (CG) method together with the hierarchical BPX preconditioner using the block Jacobi smoothing where the block structure is obtained, as above, by collecting all nodal unknowns for some geometric entity into one block. Since problem (1) is elliptic and symmetric, this is a good choice, because the hierarchical BPX preconditioner inside the CG method ensures robustness against $h$-refinement (that is, the convergence rate does not deteriorate when refining the mesh, see e.g. the monograph [10]) and using the block Jacobi smoother ensures a certain (albeit not perfect) robustness against variations in the finite element order $p$.

### 3 Performance analysis and shared-memory parallelization

#### 3.1 Performance of the serial code

Running the algorithm described above results in the approximation results shown in Table 1 for the component $\hat{A}_{11}^{11}$ of the effective tensor, where we underlined those digits for which an approximation coincides with the approximation on the next refined level. The letters K, M, G in that table denote $10^3$, $10^6$, and $10^9$, respectively.

For obtaining these values, we have used a compute server featuring 256 GB RAM and a NUMA architecture with an AMD Opteron 6180 SE CPU (4 sockets each having 12 cores and no hyperthreading) running at 2.5GHz. This server is some years old, and its serial performance is at least a factor 3 smaller both in arithmetic performance and memory bandwidth than more recent hardware. On the other hand, it is a very good platform for parallelization tests, because of the large memory and the large number of parallel operations available.

¹Like, for example, the SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions), which are available on the Intel and AMD processors.
### Table 1: Approximation of $\hat{A}_{11}^{11}$ with Lagrange FE of order 5.

<table>
<thead>
<tr>
<th>Level</th>
<th>Cells</th>
<th>Unknwons</th>
<th>Matrix entries</th>
<th>Time (s)</th>
<th>$\hat{A}_{11}^{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>22.167K</td>
<td>2.12473M</td>
<td>7</td>
<td>2.6235177043</td>
</tr>
<tr>
<td>1</td>
<td>48</td>
<td>192.024K</td>
<td>18.5712M</td>
<td>85</td>
<td>2.6231458879</td>
</tr>
<tr>
<td>2</td>
<td>384</td>
<td>1.51991M</td>
<td>148.704M</td>
<td>690</td>
<td>2.6231424466</td>
</tr>
<tr>
<td>3</td>
<td>3072</td>
<td>12.0165M</td>
<td>1.18752G</td>
<td>5540</td>
<td>2.6231424269</td>
</tr>
</tbody>
</table>

3.2 Performance analysis of the serial code

An important question now is: How fast is the optimized serial code relative to the peak performance of the machine? Which, in turn, leads us to the more fundamental question: How fast could an optimal program run on this architecture under the restriction that it still performs the same numerical algorithm?

This latter question can be answered using the well-known Roofline Model (see e.g. [16]) which determines the maximal achievable performance $P$ of an algorithm on a given computer architecture as

$$P = \min(P_{\text{max}}, b_S \cdot I),$$

where $P_{\text{max}}$ is the maximal floating point performance, $b_S$ is the achievable main memory bandwidth and the computational intensity $I = \frac{n_{\text{FLOP}}}{n_{\text{Byte}}}$ of the algorithm is the quotient of the number of floating-point operations and the amount of data which has to be moved between main memory and processor cache for the algorithm under consideration.

For our architecture, the processor data sheets suggest a floating point performance of $P_{\text{max}} = 10 \text{GFLOPS}$ (fused multiply-add and SSE at 2.5GHz) and a maximum memory bandwidth of about 10 GB/s per socket. However, we settle for the more realistic values $P_{\text{max}} = 7.5 \text{GFLOPS}$ and $b_S = 4.5 \text{GB/s}$, given that $P_{\text{max}} = 7.5 \text{GFLOPS}$ is the maximal performance which we observe for the highly-optimized BLAS-Level 3 routines GEMM and GETRF and $b_S = 4.5 \text{GB/s}$ is the maximum bandwidth we observe when running the well-known STREAM benchmark (see [20]) on our hardware.

For the computational intensity $I$ we use the estimate

$$I = \frac{n_{\text{FLOP}}}{n_{\text{Byte}}} = \frac{8.0 \cdot 10^{12} \text{FLOP}}{2.9 \cdot 10^{12} \text{Byte}} \approx 2.8 \frac{\text{FLOP}}{\text{Byte}},$$

where $8.0 \cdot 10^{12} \text{FLOP}$ is the number of floating point operations of all BLAS level 3 operations (i.e., GEMM and GETRF), which can easily be measured and is both a lower bound and a good estimate of the overall number of processing units.
floating point operations, and $2.9 \cdot 10^{12}$ Byte results by multiplying the sum of all sizes of vectors and matrices used during the solving process with the number of iterations.

Therefore, the theoretical performance limit induced by the memory bandwidth is $I \cdot b_S \approx 13$ GFLOPS, which is about twice as large as $P_{\text{max}} = 7.5$ GFLOPS. Thus, our program is not memory bound on this architecture, and the optimally achievable performance should be $P_{\text{max}} = 7.5$ GFLOPS.

Contrasting this optimal performance with the actual performance $P = \frac{4.9 \cdot 10^{12} AO}{5540 s} = 1.44$ GFLOPS which our serial program did achieve according to Table 1, we see that there is still much room for improvement. On the other hand, there are rather obvious reasons for this under-performance: first, there is an impact of the discretization on unstructured meshes with nonlinear cells which was ignored in the above analysis, and second, that the block matrices for the BLAS level 3 operations are not always large (e.g., the diagonal blocks for unknowns located at vertices are of size 3x3 only!), and that, consequently the memory overhead as well as the BLAS method call overhead—both from the Common Lisp side and from the LAPACK side—is not that negligible that 7.5 GFLOPS can be achieved. Both these problems cannot be circumvented easily, so that we accept this serial efficiency and try to obtain further speed improvements by parallelization.

### 3.3 Shared-memory parallelization

Finite Element calculations contain, in general, two prominent and time-consuming operations. Firstly the discretization of the mathematical problem and secondly the solution process for the discretized linear systems. These are also the two points where parallelization efforts are expected to yield the most benefits.

Parallelization of the discretization process in finite element methods is relatively straightforward, with one exception: Updates of the global matrix and the defect have to be synchronized between concurrent threads. In contrast, the parallelization of the linear solver is usually much more involved and sometimes even impossible. Luckily our particular solution algorithm—the BPX-preconditioned CG method—was explicitly designed with parallelization in mind.

Support for shared-memory parallelization is not contained in the Common Lisp language specification (see [23]), but all modern Common Lisp implementations provide some means of using the threading capabilities of the underlying operation system. Furthermore, the **bordeaux-threads** library (see [26]) provides a unified interface to these implementation-specific thread extensions, and the **lparallel**-library (see [19]) provides several higher-level abstractions, for example worker pools, parallel map/reduce and a controller.

---

2Measurements show that about 30% of the total execution time is spent in the assembly phase.
Table 2: Solution time in seconds depending on refinement level and number of threads measured on one socket of an AMD Opteron 6180 SE processor.

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>85</td>
<td>57</td>
<td>49</td>
<td>41</td>
<td>38</td>
<td>37</td>
<td>32</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>690</td>
<td>440</td>
<td>360</td>
<td>320</td>
<td>290</td>
<td>280</td>
<td>240</td>
<td>240</td>
</tr>
<tr>
<td>3</td>
<td>5540</td>
<td>3620</td>
<td>3020</td>
<td>2720</td>
<td>2380</td>
<td>2300</td>
<td>1960</td>
<td>1860</td>
</tr>
</tbody>
</table>

for interactive debugging of multiple threads.

The shared memory parallelization in FEMLISP uses the worker pools of parallel extending it with an additional syntactic construct

\[
(\text{with-workers} \ (\text{func})
\quad \ldots \ (\text{work-on} \ \ldots) \ \ldots)
\]

where the call \((\text{work-on} \ \ldots)\) can appear in any context and pushes its arguments on a queue, from which it is then taken and processed by the workers of the worker pool using the function parameter \text{func}. On first sight, this \text{with-workers} \ldots\text{work-on}-construct looks more restricted than the similar \text{cilk} \ldots\text{spawn} in Cilk ([24]) where arbitrary calculations can be spawned, but in fact, it is equivalently expressive, because that functionality can easily be obtained by giving \text{work-on} functional arguments.

For avoiding NUMA effects, we use only cores associated to a single socket for our next test (we will treat different sockets as different nodes in Section 4), and obtain the times shown in Table 2. We observe that we can achieve a speedup of a factor 3 on this hardware, although this speedup is only reached when employing relatively many processors. The reason for this behaviour is twofold: first, there is a synchronization overhead when possibly shared data is updated, and second, the scattered data causes a lot of cache misses in the rather complicated cache structure of the AMD Opteron. As we will see later, this can be improved on by moving from shared-memory to distributed-memory parallelization. Note also, that the concurrent access to the queue used internally by \text{with-workers} cannot account for the limited speedup, because the worker function is far too work-intensive in our application.

4 Distributed-memory parallelization

While the shared-memory parallelization is rather straightforward and required almost no changes to FEMLISP application code, distributed-memory
parallelization is a more demanding task. We attack the problem by introducing two libraries which shield the user from the intricacies of parallel hardware. The first library, cl-mpi, is a low-level interface which gives Common Lisp easy access to MPI functionality. The second, ddo, is a higher-level interface which starts from the assumption that all relevant data is stored in the form of objects and which manages the distribution and synchronization of these objects.

In the following, we describe those two libraries.

4.1 cl-mpi — A Common Lisp MPI interface

The library cl-mpi allows Common Lisp programs to use the Message-Passing Interface MPI. The library is more than a foreign function interface between the C Language and Common Lisp. Common Lisp is a powerful dynamically typed language with many convenience features and cl-mpi was written to take full advantage of these features, without sacrificing either bandwidth or latency. We will first discuss the syntax of cl-mpi and the benefits for the programmer, followed by performance measurements that compare the C Language to code written with cl-mpi.

4.1.1 Usability

A typical MPI call in the C Language is given in Figure 2, together with two similar calls in cl-mpi. The given example is simple yet illustrates the major differences. The result is clearly visible, the cl-mpi snippets are much more concise.

Common Lisp has a powerful mechanism for handling exceptional situations — the Common Lisp Condition System — which is beyond the scope of this report. We use the Condition System to gracefully handle all MPI errors “behind the scenes” without adding syntactic overhead to the individual MPI calls. We only note here, that every error is detected and signalled and a user can optionally choose to handle or even ignore the error.

The biggest gain in clarity in cl-mpi is gained by introducing default and keyword arguments. Even the simple MPI_Receive statement in Figure 2

```c
ierr = MPI_Recv(data, N, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, &status);
CHECK_FOR_ERRORS(ierr);
```

```lisp
(mpi-recev data src) ; a simple cl-mpi call
(mpi-recev data src :tag 12 :start 5 :end 9) ; more elaborate
```

Figure 2: Comparing C Language MPI with cl-mpi
has seven mandatory arguments that the programmer has to remember, including their correct order. We lowered the number of mandatory arguments in cl-mpi to only two arguments. These are the array that should be used to send or receive the data and the id of the communication partner. The datatype of the message is automatically inferred from the dynamic type of the array. The tag parameter defaults to zero, the start and end parameter default to zero and the end of the array, respectively. Furthermore all but the first two arguments are keyword arguments. Keyword arguments are named explicitly and can therefore appear in any order, taking yet another burden from the programmer.

Finally the handling of the MPI communicator in cl-mpi differs greatly from the C Language API. Common Lisp has lexically scoped variables by default, but also features dynamic variables, which retain their value for all code within the dynamic execution of a statement, e.g. everything that is deeper in the control stack. Although a communicator can be specified exactly, it defaults to the value of the dynamic variable *standard-communicator* which is initially bound to +mpi-comm-world+. This dynamic variable allows the most common use-case, which is to set the communicator for the dynamic execution of a block of code, by introducing only a single line of code.

### 4.1.2 Performance

Most users of expensive distributed computers value performance over convenience. MPI is successful in this market because it provides a very fast and low-level interface to the available networking hardware. In particular a process that passes a message assumes that the receiving process will eventually issue a receive operation with matching parameters. Errors that could only be detected by additional communication are not detected at all. Furthermore MPI allows to send or receive data from an existing array or buffer without any intermediate copies. The cl-mpi library does not deviate from this behavior and therefore attains the same bandwidth as a plain C implementation as can be seen in Figure 3.

There are some differences between the C or Fortran bindings of MPI and cl-mpi, as described in section 4.1.1. The cl-mpi notation is both shorter and less error prone, as it avoids handling raw pointers to memory. Moreover cl-mpi uses the Common Lisp Condition System to signal an exception if MPI returns a nonzero error code. These changes incur a small but measurable increase in latency, because each cl-mpi method has to do some additional conversion and checking before it issues the operation. To determine the latency overhead, we used the ping-pong benchmark as described in Figure 3. The measurements show that the latency difference is roughly 500 nanoseconds per ping-pong iteration. A single iteration consists of two send and receive pairs, so the latency overhead introduced by cl-mpi is about
Table 3: Comparing the Intel MPI benchmark suite (IMB) with cl-mpi in the pingpong benchmark for intra socket communication on a Intel i7-5500U CPU (2.4GHz) using MPICH2 version 3.1.0.3. The lisp code was executed with SBCL 1.2.4 and the C code was compiled with GCC 4.9.2 and highest optimization settings.

```lisp
(loop repeat iterations do
  (if (evenp rank)
    (let ((target-rank (+ rank 1)))
      (mpi-send buf target-rank :end msg-size)
      (mpi-recv buf target-rank :end msg-size))
    (let ((target-rank (- rank 1)))
      (mpi-recv buf target-rank :end msg-size)
      (mpi-send buf target-rank :end msg-size))))
```

Figure 3: The MPI ping-pong benchmark

250 nanoseconds per point to point communication. Given that the physical latency of modern Infiniband hardware is typically above one microsecond, we claim that the abstraction overhead of cl-mpi is indeed tolerable and well worth the added convenience.

### 4.2 Dynamic Distributed Objects

While cl-mpi is a clear improvement on the C interface of MPI, it does not offer a sufficiently high abstraction level for painless parallel programming. Ideally, parallelization should happen more or less automatically, given the underlying algorithm is parallelizable in the first place.

We make a big step towards this goal by introducing parallel data structures in an object-oriented way and a library which we call Dynamic Distributed Objects (in short: ddo). The library was inspired by the library
DDD (*Dynamic Distributed Data*) described in [4], and we discuss similarities and differences between those two approaches in more detail below.

In DDO, distributed objects have the following distinctive features:

- Distributed objects are derived from standard CLOS\(^3\) objects using multiple inheritance. I.e., every object can, in principle, become a distributed object. This offers uttermost flexibility while keeping the important advantage, that the parallelization overhead is restricted to real interface objects only.

- Distributed objects can be garbage-collected. No explicit object removal is necessary. This allows keeping the advantages of dynamic garbage-collected languages even when doing distributed calculations.

- Synchronization of distributed objects occurs only at certain synchronization points in the algorithm, keeping the communication overhead small.

- At each synchronization point, a two-pass synchronization occurs between all neighboring processors. In the first pass, the local ids of new, changed, and deleted distributed objects are communicated. In the second pass a message of all necessary data updates is compiled and sent to all affected neighbors. On each processor the function stored in the special variable *synchronization-merger* is called with two arguments: first, the distributed object, and second, the data of all identified objects in the neighboring processors.

- Depending on what is to be done during a certain synchronization step, the user can arbitrarily customize this variable in a local context. For example, there are pre-defined mergers, which compute the new value of object data by adding the data of an object and its identifications, or choose the data field from a certain worker (e.g. the one with lowest mpi-rank) as the new value.

The user interface of DDO is shown in the upper part Table 4. Although it contains only two functions and one special variable, it is nevertheless flexible due to the complete freedom in choosing the synchronization merger function.

For getting an idea of DDO’s implementation, we have listed some internal variables in the lower part of Table 4. They are used in the following way:

- The special variable *local-id-count* contains the counter which is used for generating unique local ids for distributed objects. Of course, this variable has to protected against concurrent access.

- Distributed objects are stored in the hash table *distributed-objects* as key-value pairs ⟨local-id, object⟩. This allows for retrieval of any object, given its local-id. However, *distributed-objects* is a so-called

\(^3\)CLOS is the *Common Lisp Object System*. 

11
DDO User interface

<table>
<thead>
<tr>
<th>Functions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>make-distributed-object</td>
<td>Turns an object into a distributed object</td>
</tr>
<tr>
<td>synchronize</td>
<td>Communicates new/changed/deleted distributed objects</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>synchronization-merger</em></td>
<td>Contains the merger function for object synchronization</td>
</tr>
</tbody>
</table>

DDO Internals

<table>
<thead>
<tr>
<th>Special variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><em>local-id-count</em></td>
<td>Counter for local ids</td>
</tr>
<tr>
<td><em>distributed-objects</em></td>
<td>Weak-value hash table mapping local-id to object</td>
</tr>
<tr>
<td><em>distribution</em></td>
<td>The 3-term relation $R$(local-id, processor, distant-id)</td>
</tr>
<tr>
<td><em>deleted-local-ids</em></td>
<td>A list of local-ids whose objects have been garbage collected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class mixins</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ddo-mixin</td>
<td>Class mixin for distributed objects</td>
</tr>
</tbody>
</table>

Table 4: Structure of DDO

“weak” hash-table. “Weakness” means in this context that entries in that table do not prevent the garbage collector from removing that object. Additionally, when such a distributed object is removed, the garbage collector calls a so-called “finalizer” which is a function that pushes the corresponding local id to the list *deleted-local-ids*. At the next synchronization point, all local-ids of the deleted distributed objects are removed from the relation, and their removal is communicated to all affected neighbors.

- The data structure in *distribution* implements a mathematical relation between three items: a local index, the neighboring processor, and the index of the object there. It is implemented with the help of balanced red-black binary trees which allows for fast insertion/removal of individual objects. This relation is used extensively in the synchronization process: it is populated when identifying objects, it is used for composing or reading the synchronization messages (note that the binary-tree-based indices allow to list distributed objects on a certain interface in a well-defined order), when distributed objects change or when they are garbage collected.

- Finally ddo-mixin is a CLOS class containing only a slot for the local id. This class is mixed with a standard class to form a new distributed class with two superclasses: the original class and the mixin. The distributed object is then simply upgraded\(^4\) to this new class.

\(^4\)Common Lisp provides this functionality with the generic function `change-class`. 

12
Let us comment a bit on the connection of DDO to other work. As already stated above, DDO is inspired by the library DDD ([4], [9]) which was fundamental for parallelizing the PDE solver UG (see [5], [6]). Recently, a successor of DDD, named PCL, was presented in [25]. The main differences between DDD, PCL, and DDO are the following:

- Both DDO and PCL work on objects, while DDD as a C library uses a more rudimentary description of data.
- DDD is implemented using global indices, while PCL and DDO maintain a consistent relation between local indices.
- The deletion of objects happens automatically in DDO, and must be performed manually in DDD and PCL.
- The user interface is much simpler for DDO than for DDD or PCL. Especially, at the moment there is no explicit functionality for migrating objects to other processors. However, a transfer of objects from processor A to B can be obtained by creating copies on B, identifying them with the originals on A, synchronizing, and then dropping the originals on A.

Apart from DDD and PCL, there are also many other approaches to distributed data. Some of them (e.g. CORBA, DCOM, . . . ) use a fine-granular synchronization protocol by remote procedure calls which is not efficient enough in an HPC context. From those which support bulk synchronization, the most popular is probably PGAS (partitioned global address space) which is the underlying parallel model for languages like UPC, Co-Array-Fortran, Chapel, X10, and others (cf. [11]). However, comparing PGAS languages with DDO, we see fundamental differences: DDO works on an object level and integrates its entire functionality within the OOP framework of a dynamic language. Existing tools for manipulation and introspection can be reused and different utilities can be combined in a homogeneous way. In contrast, PGAS languages work on a lower level of abstraction and hardly fit into the semantics of any existing programming language. Thus new languages are frequently designed explicitly around the PGAS model. Needless to say, reuse of components and collaboration is mostly lost this way.

4.3 Distributed-memory parallelization of Femlisp

Interestingly, distributed-memory parallelization of the benchmark problem could be added to FEMLISP almost completely as an add-on library with only minimal modifications of the original (thread-parallelized) sequential code. More precisely, only the following changes in functionality were necessary:

1. For mesh management, we introduced the possibility to distribute a mesh. At the moment, this is done in the following way: All processors start from a common coarse mesh. The cells in those coarse meshes are
then identified with the corresponding ones belonging to other processors. Finally, the parts of the mesh which should belong only to other processors are simply dropped (and left to the built-in garbage collector and DDO). The result is a mesh where only an interface consisting of lower-dimensional entities (faces, edges, vertices of cells) is distributed.\footnote{At least, this is how it is done at the moment. There are other algorithms, for which more overlap is needed.}

2. Mesh refinement is augmented by ensuring that the children of distributed cells become distributed objects themselves which are correctly identified with the corresponding ones on the neighboring processors.

3. Discretization works in parallel almost without changes, because we decided to work with processor-local stiffness-matrix and processor-local load-vector. Therefore, this step does not need synchronization, and its result are the so-called \textit{inconsistent} matrices $A_i$ and vectors $f_i$ following the terminology of \cite{5} and \cite{2}.

4. The solver step requires more modification. For easy evaluation, the approximate solution should be globally consistent (we denote this by $u_c$), so consistency has to be introduced at some place. Indeed, it turns out that only three places have to be modified:

- The (block) diagonal entries of the stiffness matrix which are needed in the Jacobi smoothing step are not yet available at the interface. We therefore generate a consistent diagonal matrix $D_c$ by adding up the inconsistent diagonal part of $A_i$ at the interfaces.

- Multiplication of the inconsistent matrix $A_i$ with the consistent approximate solution $u_c$ yields an inconsistent vector $A_iu_c$. Therefore, also the residual obtained as $r_i = f_i - A_iu_c$ is inconsistent. The correction vector calculated by the Jacobi smoothing step as $c_i = D_c^{-1}u_i$ is again inconsistent, such that it has to be made consistent $c_c = S_{I \rightarrow C}c_i$ before adding it to the consistent solution $u_c$. This operator $S_{I \rightarrow C}$ involves communication, because it adds up all inconsistent components and distributes the resulting sum.

- For monitoring the progress of the iteration, we track the Euclidean norm of the residual vector. For debugging purposes, it is important that this number is available, so we have to calculate the Euclidean norm of the \textit{consistent} residual. We do this by first calculating $r_c = S_{I \rightarrow C}r_i$ and then performing the calculation $\|r\|_2^2 = \langle r_c, r_i \rangle$. Note that only the first operation requires exchanging a significant amount of data, whereas in the calculation of the scalar product only scalars have to be exchanged.

The final result is a code which runs in identical form on a serial machine, a shared-memory parallel machine, and on a distributed-memory parallel cluster.
For the end user, the main difference lies in the kind of interactive use. In the serial and thread-parallel setting, there is only one Common Lisp instance with which (s)he can directly interact. In contrast, the CL instances running on cluster will usually either work in batch mode or will be controlled by an exterior client. Of course, for development, the second variant is much more interesting. We have found it most convenient to use as client another CL instance which runs either on the master node of the cluster (it needs only little memory and CPU time) or on a remote workstation. This client does not really need FEMLISP or DDO, but having those libraries installed is convenient, because it allows for easy interactive control (using the library LFARM, see [18]) without having to worry about undefined symbols.

4.4 Numerical results for distributed-memory parallelization

For having more possibilities of distributing the load evenly, we start from level 1 with 48 cells as the base-mesh. Then we repeat the calculation from Tables 1 and 2 employing more than one MPI worker. For using all processors of the machine, each MPI worker uses \( \min\{12, \frac{48}{\#\text{workers}}\} \) threads. We then obtain Table 5. We see that we can obtain an additional speedup of more than a factor 6 compared with the shared-memory parallel results from Table 2.

Finally, we do the same calculation on the HPC cluster “LiMa” (Little Machine) of the computing center RRZE at the FAU which is equipped with stronger Intel Xeon X5650 Westmere CPUs operating at 2.66 GHz. The batch system of this machine only allows to allocate entire nodes, and because each of the nodes of this machine has two sockets, we choose the number of MPI workers such that each of them works on an own socket. We then obtain the compute times from Table 6, from which we can then compute the parallel efficiency in Figure 4. Note that we have used the times for \( p_0 = 2 \) workers on levels 2 and for \( p_0 = 6 \) workers on level 3 as reference values for calculating the parallel efficiency \( E \) using the formula

\[
E(p) = \frac{p_0}{p} \frac{T(p_0)}{T(p)}
\]

(here \( T(p) \) denotes the solution time measured for \( p \) workers), because these levels do not fit into the memory of a single socket.

<table>
<thead>
<tr>
<th>Cells</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>42</td>
<td>25</td>
<td>17</td>
<td>13</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>17</td>
</tr>
<tr>
<td>384</td>
<td>300</td>
<td>170</td>
<td>105</td>
<td>80</td>
<td>74</td>
<td>77</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>3072</td>
<td>1985</td>
<td>960</td>
<td>565</td>
<td>370</td>
<td>330</td>
<td>330</td>
<td>310</td>
<td>305</td>
</tr>
</tbody>
</table>

Table 5: Solution time in seconds depending on refinement level and number of MPI workers.
Table 6: Solution time in seconds on the LiMa cluster.

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>20</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>100</td>
<td>58</td>
<td>45</td>
<td>36</td>
<td>27</td>
<td>24</td>
<td>23</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>—</td>
<td>240</td>
<td>190</td>
<td>130</td>
<td>105</td>
<td>82</td>
<td>53</td>
<td>—</td>
</tr>
</tbody>
</table>

Figure 4: Parallel efficiency on the LiMa cluster.

Now, having the cluster at our disposal, it would be possible to refine the mesh once more. However, this does not really make sense from a practical point of view, because roundoff errors in the calculation of the stiffness matrices and residuals together with the condition of the stiffness matrix limit the accuracy to about nine digits.

5 Discussion and outlook

When we compare the 5540 seconds for computing the effective coefficients with the 305 seconds from Table 5, we can see that we were able to obtain by parallelization a speedup of 27 on a compute server, and the same speedup could also be observed on a cluster. This became possible due to shared-memory parallelization giving us a speedup of more than 3 and distributed-memory parallelization resulting (for this relatively small problem) in an additional speedup of more than 6.

Although the total speedup is quite satisfactory, the largest advancement is that the capabilities of our library Femlisp are now extended from model problems to much larger real-world problems, simply because we are not limited any more by software capabilities but only by the availability of suitably large computing clusters. Thus, Femlisp should be able now to solve in-
teractively rather general partial differential equations posed on complicated physical domains.

An important point is also, that this progress was possible in a very un-intrusive way. More precisely, the shared-memory parallelization happens automatically as soon as a machine with OS threads is available\(^6\), and even the distributed-memory parallelization could be implemented as a rather small\(^7\) add-on library to the serial (resp. shared-memory parallel) FEMLISP code mainly thanks to the conveniences of Common Lisp and our libraries cl-mpi and DDO.

It is true, however, that the current state of FEMLISP parallelization is still far from complete. First, there are some parts like the calculation of quadrature rules, shape functions, etc, which are still serial, even if they would not have to be. But, second and much more important, our distributed-memory parallelization is currently based on a static domain-decomposition, which has the drawback that adaptive mesh refinement would rapidly introduce large load imbalances. To overcome this problem, a more flexible scheme, like, for example, the one described in [6], [3] which allows for the transfer of cell clusters in the mesh hierarchy becomes a necessity. The flexible approach of FEMLISP/DDO will surely help a lot in achieving this ultimate form of parallelized adaptivity, but at the moment this has not yet been done.

Acknowledgements

- This work was supported by the Kompetenznetzwerk für wissenschaftliches Höchstleistungsrechnen in Bayern (KONWIHR III Förderprogramm “Simulationssoftware für Multi-/Manycore-Systeme”).
- The authors want to thank Eberhard Bänisch, Pascal Costanza, and Balthasar Reuter for useful discussions.
- The code used for the tests in this article can be obtained following the installation instructions given on the FEMLISP Homepage [14]. Moreover, when running FEMLISP most of the calculations of this article can be performed easily inside an interactive demo.

References


---

\(^6\)At least, this is the case for the discretization, whereas not every solver is parallelizable.

\(^7\)More precisely: The DDO-FEMLISP interface consists of 600 lines of code (LOC). It uses the general-purpose libraries DDO (775 LOC) and cl-mpi (1700 LOC). This has to be compared with the size of FEMLISP (43000 LOC).


Zuletzt erschienene Beiträge:

386 N. Ray, P. Knabner: Upscaling flow and transport in an evolving porous medium with general interaction potentials (06.10.2014)
387 J. Greifenstein, M. Stingl: Simultaneous material and topology optimization (21.10.2014)
388 R. Schulz, N. Ray, F. Frank, H. Mahato, P. Knabner: Strong solvability up to clogging of an effective diffusion-precipitation model in an evolving porous medium (06.02.2015)
389 E. Bänisch, A. Brenner: A-posteriori estimates for the rotational pressure-correction projection method (29.05.2015)
390 M. Bischoff, J. Jahn: Economic objectives, uncertainties and decision making in the energy sector (07.10.2015)
391 R. Schulz, P. Knabner: Derivation and analysis of an effective model for biofilm growth in evolving porous media (08.03.2016)
392 M. Gahn, M. Neuss-Radu: A characterization of relatively compact sets in $L^p(\Omega, B)$ (29.04.2016)
394 J. Hoffmann, S. Kräutle, P. Knabner: Existence and uniqueness of a global solution for reactive transport with mineral precipitation-dissolution and aquatic reactions in porous media (15.11.2016)
396 M. Gahn, M. Neuss-Radu, P. Knabner: Derivation of effective transmission conditions for domains separated by a membrane for different scaling of membrane diffusivity (22.03.2017)
397 M. Gahn, M. Neuss-Radu, P. Knabner: Effective transmission conditions for processes through thin heterogeneous layers with nonlinear transmission at the bulk-layer interface (24.03.2017)
398 N. Ray, R. Schulz: Derivation of an effective dispersion model for electroosmotic flow involving free boundaries in a thin strip (31.05.2017)
399 A. Rupp, K.U. Totsche, A. Prechtel, N. Ray: Discrete-continuum multiphase model for structure formation in soils including electrostatic effects (28.05.2018)
402 S. Kräutle, J. Hodai, P. Knabner: Robust simulation of mineral precipitation-dissolution problems with variable mineral surface area (04.03.2019)
403 R. Schulz, N. Ray, S. Zech, A. Rupp, P. Knabner: Beyond Kozeny-Carman: Predicting the permeability in porous media (20.03.2019)
405 M. Heisig, N. Neuss: Optimization and distributed-memory parallelization of the finite element library Femlisp (15.04.2019)