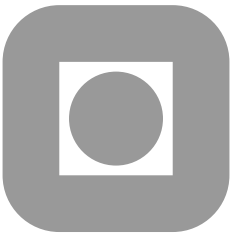
NORGES TEKNISK-NATURVITENSKAPELIGE
UNIVERSITET

A fast tensor-product solver for incompressible fluid flow in partially deformed three-dimensional domains: Parallel implementation

by

Arne Morten Kvarving, Einar M. Rønquist*

PREPRINT
NUMERICS NO. 10/2010



NORWEGIAN UNIVERSITY OF
SCIENCE AND TECHNOLOGY
TRONDHEIM, NORWAY

This report has URL
http://www.math.ntnu.no/preprint/numerics/2010/N10-2010.pdf
Address: Department of Mathematical Sciences, Norwegian University of Science and Technology, N-7491 Trondheim, Norway.

A fast tensor-product solver for incompressible fluid flow in partially deformed three-dimensional domains: Parallel implementation

Arne Morten Kvarving, Einar M. Rønquist*

October 7, 2010

We describe a parallel implementation of the tensor-product solver derived in [5, 29]. A combined distributed/shared memory model is chosen, since the flexibility allows us to map the algorithm better to the available resources. Since the approach requires special attention to load balancing, we also propose a scheme that resolves the challenges involved. Speedup results from test problems, as well as from real simulations, are presented and discussed. While the speedups are not perfect, we show that the new algorithms are more than competitive with a standard 3D approach parallelized using domain decomposition.

Keywords: tensor-product solver, shared memory, distributed memory, Bénard-Marangoni

1 Introduction

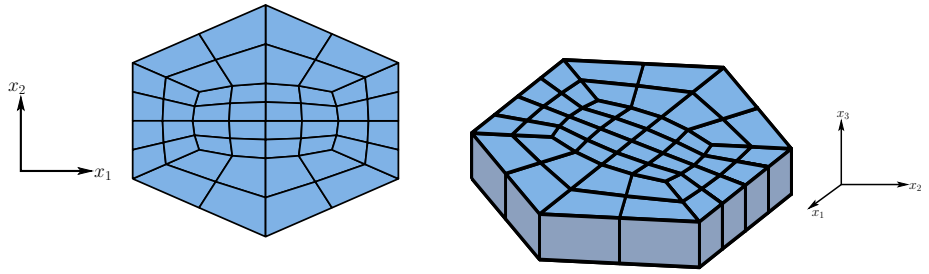
Parallelization strategies for finite element codes are usually based on the domain decomposition paradigm [12, 30]. This way of extracting parallelism has many attractive features, such as the availability of very efficient solvers, relatively easy implementation and access to load balanced codes. In principle, all that is required is a sufficient number of elements compared to the number of processors, as well as a good division of these elements between the processors. This gives a coarse grained division of the workload which maps very well to a distributed memory model where separate processes communicate through message passing.

In this document we study parallelization strategies for another class of algorithms, which allows for alternative approaches. These algorithms are only applicable to a certain class of problems, namely problems in geometries of the "cylindrical" kind which can be viewed as an extrusion of some general 2D cross-section [5, 10, 29]; see Figure 1 for an example. Our motivation for considering this particular class of geometries is that we want to use the code to simulate surface-tension-driven Bénard-Marangoni convection in confined containers [7, 8, 9, 16, 17, 22, 28]. This class of algorithms decompose the elliptic 3D problems into a set of completely decoupled 2D problems. The decoupling into several subproblems offers a new parallelization strategy; instead of having all processors participate in the solution of one large problem, we can now divide them into groups which work independently of each other. In other words, the parallelization strategy is to a large extent given by

the algorithm. The division of the processors into groups means that we have to face additional challenges when it comes to load balancing, since the different groups may have varying solution times. Nonetheless, these algorithms are interesting since from experience [5, 29] they reduce the number of floating point operations needed by close to an order in magnitude and are very conservative with respect to memory usage. When applicable, they should reduce the amount of computing resources needed significantly.

In terms of spatial discretization, the focus in this work is on spectral elements. We remark this does not reflect a limitation of the algorithms considered. The algorithms can be used with any spatial discretization, such as low-order finite elements or finite differences. None of the following discussion relies directly on the use of spectral elements, thus the parallelization derived is also applicable with these kinds of spatial discretizations. For instance, earlier work has been reported where a Fourier expansion in the extrusion direction is combined with an element method [11, 23, 25]. Our approach can certainly be applied in this case.

The outline of the paper is as follows. In Section 2 the description of the (model) problem considered is given. In Section 3 we briefly discuss the discretization leading to the linear systems of equations we have to solve. In Section 4 we give a brief overview of the tensor-product solvers considered, before moving on to discuss the parallelization in Section 5. Speedup results are then given in Section 6. Finally, in Section 7, we summarize our findings and present our conclusions.



(a) cross-section (b) 3D geometry

Figure 1: An example of the class of geometries we consider. Here, the cross-section is a hexagon which is extruded in the third direction to form a full 3D container.

2 Problem definition

As a model problem, we consider the unsteady Stokes equations in some (extruded) domain Ω;

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} - \nabla^2 \mathbf{u} + \nabla p &= \mathbf{f} \quad \text{in } \Omega, \\ \nabla \cdot \mathbf{u} &= 0 \quad \text{in } \Omega, \end{aligned} \tag{1}$$

where $\mathbf{u}$ is the velocity, p is the pressure and $\mathbf{f}$ represents a given body force. We assume that appropriate initial and boundary conditions for the velocity are specified. For simplicity, we here assume that

$$\mathbf{u} = \mathbf{0} \quad \text{on } \partial\Omega.$$

The governing equations for incompressible fluid flows are the Navier-Stokes equations. However, since the convection operator is typically handled using explicit time integrators

(following a semi-implicit approach), this does not give rise to additional elliptic systems. Thus, the Stokes equations serve us well, since we here consider the solution of the elliptic systems of equations derived from an implicit-in-time method in combination with a velocity-pressure splitting scheme.

3 Discretization

We use high order spectral elements [20] to discretize in space, specifically the $\mathbb{P}_N/\mathbb{P}_{N-2}$ method [21]. The domain Ω is decomposed into K spectral elements, each with polynomial degree N. These elements are layered in the extrusion direction, thus K can be expressed in the form $K = \mathcal{K} \cdot \mathcal{L}$, where $\mathcal{K}$ is the number of spectral elements in each layer, and $\mathcal{L}$ is the total number of layers. We refer to a specific element using two indices: $k_i^j, i = 1, \cdots, \mathcal{K}, j = 1, \cdots, \mathcal{L}$. Here i is the element number within each layer and j the layer. These elements give rise to a number of planes, one per degree of freedom in the extrusion direction. The number of such planes depends on the boundary conditions considered, as well as N, the polynomial order of the elements. For the homogenous Dirichlet boundary conditions considered here, each velocity component will have $\mathcal{N}_1 = \mathcal{L}N - 1$ planes. Likewise, for the pressure we have $\mathcal{N}_2 = \mathcal{L}(N-1)$ planes; this would be the same no matter which boundary conditions are enforced on the velocity.

In the following, certain operations will take place across the entire span of the extrusion direction. We thus introduce a set of "super-elements", $\mathcal{E}_i = \left\{k_i^j\right\}_{j=1}^{\mathcal{L}}, i = 1, \cdots, \mathcal{K}$. Each super-element $\mathcal{E}_i$ consists of the composition of $\mathcal{L}$ spectral elements in the extrusion direction, i.e., we have $\mathcal{K}$ such super-elements.

The system of semi-discrete equations corresponding to (1) (discrete in space and continuous in time) can be expressed as

$$\begin{aligned} \mathbf{B}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} + \mathbf{A}\mathbf{u} - \mathbf{D}^T p &= \mathbf{B}\mathbf{f} \\ \mathbf{D}\mathbf{u} &= 0. \end{aligned} \tag{2}$$

Here, $\mathbf{A}$ is the discrete viscous operator (vector Laplacian), $\mathbf{B}$ the mass matrix, $\mathbf{D}$ and $\mathbf{D}^T$ represent the discrete divergence and gradient operator, respectively, while $\mathbf{u}$ and p are the unknown, nodal velocity and pressure values, respectively. Note that we use the same notation for the velocity, pressure and source in (1) and (2). In the following the symbols refer to the discrete quantities.

For clarity of presentation we here consider a first order temporal discretization. We employ a velocity-pressure splitting scheme, specifically an incremental pressure-correction scheme [3, 13, 14, 15, 26, 27]. This is done to avoid a costly coupled solution strategy, as well as the nested iterations associated with a Uzawa decoupling [6, 31]. These splitting schemes are closely coupled to the temporal discretization; in particular, they are based on backward differencing. This means that for a first order realization, we use the backward Euler method and our fully discrete problem reads

$$\begin{aligned} \frac{1}{\Delta t}\mathbf{B}\left(\hat{\mathbf{u}}^{n+1} - \mathbf{u}^n\right) + \mathbf{A}\hat{\mathbf{u}}^{n+1} - \mathbf{D}^T p^n &= \mathbf{B}\mathbf{f}^{n+1}, \\ \frac{1}{\Delta t}\left(\mathbf{u}^{n+1} - \hat{\mathbf{u}}^{n+1}\right) - \mathbf{D}^T\left(p^{n+1} - p^n\right) &= 0, \\ \mathbf{D}\mathbf{u}^{n+1} &= 0, \end{aligned} \tag{3}$$

resulting in decoupled problems for the velocity and the pressure

$$\mathbf{H}\hat{\mathbf{u}}^{n+1} = \frac{1}{\Delta t}\mathbf{B}\mathbf{u}^n + \mathbf{D}^T p^n + \mathbf{B}\mathbf{f}^{n+1}, \tag{4}$$

$$\underbrace{\mathbf{D}\mathbf{B}^{-1}\mathbf{D}^T}_{=\mathbf{E}}\Delta p^{n+1} = \mathbf{E}\Delta p^{n+1} = -\frac{1}{\Delta t}\mathbf{D}\hat{\mathbf{u}}^{n+1}. \tag{5}$$

Here Δt is the time step, superscript n refers to the quantity evaluated at time $t^n = n\Delta t$, $\mathbf{H} = \mathbf{A} + \frac{1}{\Delta t}\mathbf{B}$ is the discrete Helmholtz operator, $\mathbf{E}$ is the consistent Poisson operator and $\Delta p^{n+1} = p^{n+1} - p^n$. After we have solved these equations, the nodal values for the pressure and velocity are updated through

$$\begin{aligned} \mathbf{u}^{n+1} &= \hat{\mathbf{u}}^{n+1} + \Delta t\mathbf{B}^{-1}\mathbf{D}^T\Delta p^{n+1}, \\ p^{n+1} &= p^n + \Delta p^{n+1}. \end{aligned} \tag{6}$$

4 Tensor-product algorithms

The key observation behind the new algorithms is the fact that the extruded geometries lead to tensor-product forms for the elliptic operators (Helmholtz and consistent Poisson). Specifically, we have

$$\mathbf{H} = \mathbf{B}^{1D} \otimes \mathbf{A}^{2D} + \mathbf{A}^{1D} \otimes \mathbf{B}^{2D} + \frac{1}{\Delta t}\mathbf{B}^{1D} \otimes \mathbf{B}^{2D}.$$

Here the superscripts ^{1D} and ^{2D} refer to the one-dimensional and the two-dimensional operators, respectively, i.e., all the geometry deformations are contained in the operators with the ^{2D} superscript. Likewise, the consistent pressure operator can be expressed as

$$\mathbf{E} = \mathbf{B}_*^{1D} \otimes \mathbf{E}^{2D} + \mathbf{E}^{1D} \otimes \mathbf{B}_*^{2D},$$

where

$$\mathbf{B}_* = \mathbf{B}_{up}\mathbf{B}^{-1}\mathbf{B}_{up}^T$$

is a special pressure operator constructed from the rectangular matrix $\mathbf{B}_{up}$ and the velocity mass matrix $\mathbf{B}$. The $\mathbf{B}_{up}$ operator is similar to the discrete divergence operator $\mathbf{D}$ in the sense of being constructed from both the velocity and pressure basis functions. However, they differ in the sense that $\mathbf{D}$ involves differentation while $\mathbf{B}_{up}$ does not. Hence, the pressure operator $\mathbf{B}_*$ is analogous to the consistent Poisson operator $\mathbf{E} = \mathbf{D}\mathbf{B}^{-1}\mathbf{D}^T$ without differentation in the gradient and divergence operators.

Algorithm 1 Fast tensor-product solver for the Helmholtz system

$$(\mathbf{A} + \alpha\mathbf{B})\, u = g$$

in partially deformed three-dimensional geometries.

Initialization Solve the generalized eigenvalue problem of dimension $\mathcal{N}_1$

$$\mathbf{A}^{1D}\mathbf{Q} = \mathbf{B}^{1D}\mathbf{Q}\mathbf{\Lambda}$$

scaled such that

$$\mathbf{B}^{1D} = \mathbf{Q}^{-T}\mathbf{Q}^{-1}$$
$$\mathbf{A}^{1D} = \mathbf{Q}^{-T}\mathbf{\Lambda}\mathbf{Q}^{-1}.$$

Then

1) Compute
$$\tilde{g} = \mathbf{Q}^T g.$$

2) Solve
$$\left(\mathbf{A}^{2D} + (\lambda_j + \alpha)\mathbf{B}^{2D}\right)\tilde{u}_j = \tilde{g}_j \qquad \forall\ j = 1, \cdots, \mathcal{N}_1.$$

3) Finally compute the solution as
$$u = \tilde{u}\mathbf{Q}.$$

The idea is now to consider appropriate eigenvalue problems, much like in standard FDM methods [18]. This allows us to decouple the solution of the 3D problem into the solution of a set of independent 2D problems. The details can be found in [5, 29]. The two algorithms result in the two solution strategies given in Algorithm 1 and Algorithm 2. Note that these algorithms are written for data stored in a particular layout. The data is indexed using two indices; one global number within each cross-section, and then a separate index to indicate which plane in the extrusion direction. Thus the vectors involved are not vectors in the traditional sense, they are entities stored in this mixed global-local layout. This means that the first and last step of the algorithms can be performed as one matrix-matrix product per super-element $\mathcal{E}_i, i = 1, \cdots, \mathcal{K}$.

At each time level, we first solve (4) using Algorithm 1 for each component to obtain $\hat{\mathbf{u}}^{n+1}$. We then solve (5) using Algorithm 2 to obtain Δp^{n+1}. Finally we update the velocity and pressure according to (6). We now move on to discuss how to parallelize Algorithms 1-2.

5 Parallelization

The algorithms dictate some constraints on our parallelization, in the sense that we have to deal with the fact that we have several independent 2D problems to solve. This in contrast to a standard 3D approach where the challenge is how to make many computing elements collaborate in the solution of a single problem. However, the algorithms also leave substantial freedom, in the sense that we are completely free to parallelize the 2D problems as we see fit. In this section we discuss different options we have in an attempt to find the optimal parallelization strategy. In the following we assume a machine architecture where a mixed programming model is possible. This is the normal case these days, since

CPUs in general consist of multiple cores integrated in one chip, on supercomputers as well as clusters alike. While programming such machines exclusively using message passing is certainly possible, taking advantage of the convenience shared memory programming offers often lessens the load on the programmer and allows for a parallel code that is much closer to its serial equivalent. Additionally, for our problem we are not memory bound, and thus only need to optimize for performance.

We now proceed by discussing the three steps in Algorithms 1 and 2 to find the optimal parallelization strategy under these assumptions. First we need to define some nomenclature. We use the Message Passing Interface (`MPI`) library [1] when programming using a distributed memory programming model. We refer to the separate `MPI` processes as *nodes*. Likewise, we use the `OpenMP` [2] standard for programming using a shared memory programming model, and refer to the individual processing cores utilized here as a *thread*. Finally, an individual 2D problem in Step 2 of Algorithm 1-2, is simply referred to as a *subproblem*.

Algorithm 2 Fast tensor-product solver for the pressure system

$$\mathbf{E}\Delta p = g$$

in partially deformed three-dimensional geometries.

Initialization Solve the generalized eigenvalue problem of dimension $\mathcal{N}_2$

$$\mathbf{E}^{1D}\mathbf{Q} = \mathbf{B}_*^{1D}\mathbf{Q}\mathbf{\Lambda}$$

scaled such that

$$\begin{aligned}\mathbf{B}_*^{1D} &= \mathbf{Q}^{-T}\mathbf{Q}^{-1}\\ \mathbf{E}^{1D} &= \mathbf{Q}^{-T}\mathbf{\Lambda}\mathbf{Q}^{-1}.\end{aligned}$$

Then

1) Compute
$$\tilde{g} = \mathbf{Q}^T g.$$

2) Solve
$$\left(\mathbf{E}^{2D} + \lambda_j \mathbf{B}_*^{2D}\right)\tilde{p}_j = \tilde{g}_j \qquad \forall\ j = 1, \cdots, \mathcal{N}_2.$$

3) Finally compute the solution as
$$p = \tilde{p}\mathbf{Q}.$$

5.1 Operating with the eigenmatrix

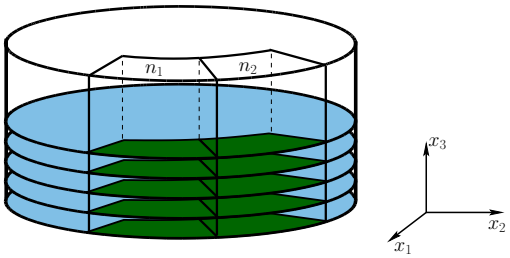


Figure 2: An example of a decomposition suitable for Step 1. Each node is assigned a set of super-elements, i.e., a domain decomposition is applied across the set of super-elements.

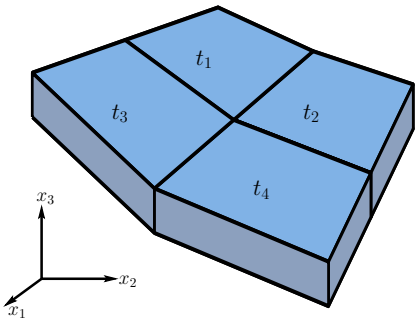


Figure 3: An example of how we would map threads to super-elements for Step 1. Each node is responsible for a certain number of super-elements, in this case 4 super-elements. The node utilizes the available threads, in this case 4, to do the operator evaluations completely in parallel. While the number of threads and super-elements match in this illustration, each thread would most likely handle several super-elements in a realistic problem.

The first step in the algorithms is to perform an operator evaluation in the x_3-direction, i.e., to perform a matrix-matrix product within each super-element. The size of the super-elemental operators depends on the grid considered. If the grid consists of many layers of elements in the x_3-direction ($\mathcal{L} >> 1$), these may grow large. In this case, these evaluations would have to be done using parallel matrix-matrix multiplications. On modern hardware, parallel matrix-matrix products should not be necessary for decompositions with less than $\mathcal{O}(1000)$ grid points in the extrusion direction. These are quite large problems; for instance this corresponds to $\mathcal{O}(10^9)$ grid points for a uniform grid. In the following we consider grids where we either have a single layer of spectral elements in the x_3-direction ($\mathcal{L} = 1$) or ten layers of spectral elements ($\mathcal{L} = 10$). In these cases the evaluations are quite small

which means parallelization within each matrix-matrix product would be totally dominated by the overhead associated with communication. However, the evaluation of the super-elemental operators can be performed in parallel, since they are completely independent of each other. As far as the optimal data distribution is concerned, this means that each node should hold a certain number of super-elements; see Figure 2. This corresponds to an "extruded" 2D domain decomposition. A realistic division would be to assign a certain number of super-elements per node. This decomposition maps very well to a combined programming model. We use threads to do the operator evaluations in the x_3-direction completely decoupled, i.e., we map super-elements to threads; see Figure 3. Ideally, each node should be responsible for exactly as many super-elements as it has threads available. In practice, we need each thread to be responsible for several super-elements to maintain high parallel efficiency, due to the overhead of thread dispatchment.

Since this decomposition basically corresponds to a classical domain decomposition, it also works well outside the elliptic solvers. This is a important, since a typical fluid flow code will entail several other steps (such as explicit treatment of the convection operator). The decomposition should result in load balanced codes for these operations as well, without any data reshuffling being necessary. We now move on to discuss the second step.

5.2 Solving the subproblems

We here assume the same domain decomposition as used in the first step. Since it is of interest to avoid any data shuffling if possible, we first investigate our options under the assumption that no data reshuffling is to be performed.

The available computing resources map nicely to the problem. While each node has data which belong to all the subproblems, this is not a problem. The nodes map a thread to each subproblem, and the individual subproblems are solved by a team of threads consisting of one thread from each node; see Figure 4. This necessitates keeping track of which `MPI` messages goes to which thread on which node. Fortunately `MPI` has built-in support for tagging messages, so a simple numbering scheme based on subproblem number and edge number can be devised. Since each thread only participates in one subproblem, they would then know which messages are meant for them. This requires a thread safe implementation of the `MPI` libraries, which most vendors do supply on modern platforms.

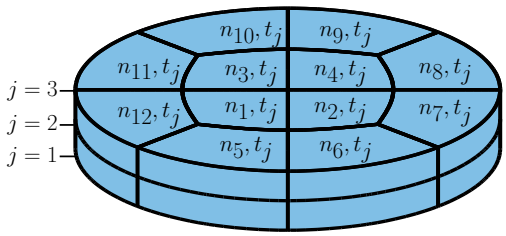


Figure 4: An example of a domain decomposition if we use combined `MPI`/`OpenMP` without performing any data reshuffling. Each node (here, 12 nodes) has spawned one thread per subproblem (here, $j = 3$ threads) and the threads corresponding to a single subproblem communicate through message passing.

In theory, this looks very nice, and would work very well in a perfect computing environment. Unfortunately, experience shows that it does not work well in practice. We run into problems due to the inherent syncronization mechanisms for threads. Since threads use shared memory, and hence shared resources, mutual exclusion locks are utilized to protect these shared resources; see Figure 5. In particular, we need to use a thread safe `MPI` library, which means that it uses a lock around its *MPI_Send()* command, in order to protect its send buffers and internal state. In computations where the actual compute nodes run on a shared memory computer, the *MPI_Send()* command is basically just a very fast local copy operation. The time spent inside the critical section is so brief that even if another caller has to wait the whole execution time, it incurs only a slight penalty (the computation time is much larger). However, this turns out to be catastrophic once we have to hit the network to send data *between* nodes. Since the code inside the critical section now takes a significant time to execute, any thread which wants to enter the code section while a transfer is in progress, have to wait for the entire transfer to finish before they can obtain the lock. This leads to a classical resource hammering problem. Even with a moderate number of threads, each thread now spends a significant amount of time simply waiting to obtain the lock, time that should be spent computing. This problem actually grows worse with both larger problem sizes (more iterations $\rightarrow$ more data exchanges $\rightarrow$ more waiting) and for more computing resources (more threads $\rightarrow$ more threads "fighting" to obtain the same lock).

In conlusion this strategy only yields good results if we restrict ourself to a single shared memory machine. In that case, using a combined programming model does not offer anything compared to a pure shared memory model. Since the motivation for considering a combined model is to allow us to use more computing resources than what can be offered on a single shared memory machine, we have no other choice than to allow for a data shuffling after Step 1 is performed.

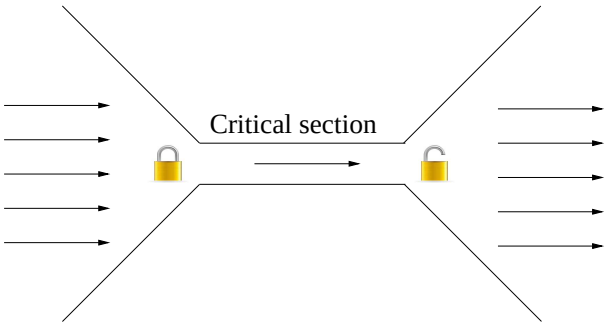


Figure 5: Illustration of a critical section. The incoming arrows represents the threads. All the threads are running concurrently, until they need to enter the critical section. Since only one thread can be inside the critical section at any time, the thread which wants to enter need to wait until it obtains the key to the mutual exclusion lock, and hence its turn to enter the code section. Once it has obtained the key and sent its message, it can start computing again. If a thread spend considerable time inside this critical section, in the worst case, the other threads all complete their computations in the same amount of time and end up waiting in the queue. In effect, the critical section almost leads to a serialization of the entire code where hardly any computations overlap since the threads spend all their time waiting in the queue.

An alternative approach is to use an all-to-all type communication to reorganize data in a more suitable manner. We collect all the data corresponding to a number of whole subproblems on each node. These subproblems can then be solved completely decoupled without any communcation. The available threads can be utilized in two ways. We can map each subproblem to a thread. This means that all the subproblems available on a node are solved without any syncronization between the threads being necessary; see Figure 6. If the number of subproblems per node is equal to the number of available threads, this seems to be the best choice. An alternative is to have all threads available on a node participate in the solution of each subproblem. This means we would utilize the threads to do operator evaluations in parallel, i.e., we apply a 2D domain decomposition method across the threads available to a node; see Figure 7.

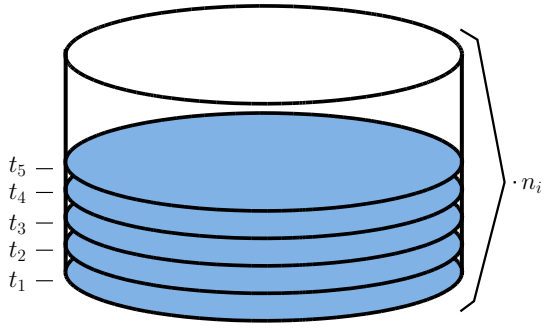


Figure 6: An example of a problem division if we use a combined `MPI`/`OpenMP` approach with data reshuffling. Nodes are handling a number of whole subproblems. The nodes then map a whole subproblem to a single thread.

In the applications we consider, the number of threads can easily be larger than the number of subproblems per node, which makes it tempting to consider the second flavor of this strategy, namely utilizing the threads to do operator evaluations in parallel. However, this does not work well in practice. We do substantially more fork/joins using this approach, and the performance gained by doing the computations in parallel is completely dominated by the overhead of thread dispatchment. Even for a relatively large problem with $K = 768$ spectral elements of high polynomial order the speedup is mediocre. If each cross-section consisted of thousands of elements, which it would typically do if we used other types of spatial discretizations, such as low-order finite elements, this might work well, but there are no guarantees. In general, we depend on the amount of work available to each thread being sufficiently large, large enough that the overhead of thread dispatchment is insignificant compared to the computation time. Using our spectral element grids, however, we only have a moderate amount of work per thread, and we hence settle for the first strategy, where each thread solves a subproblem completely on its own. This brings an upper limit on the amount of threads there is a gain from allocating on each node, in particular we cannot utilize more threads than we have subproblems available.

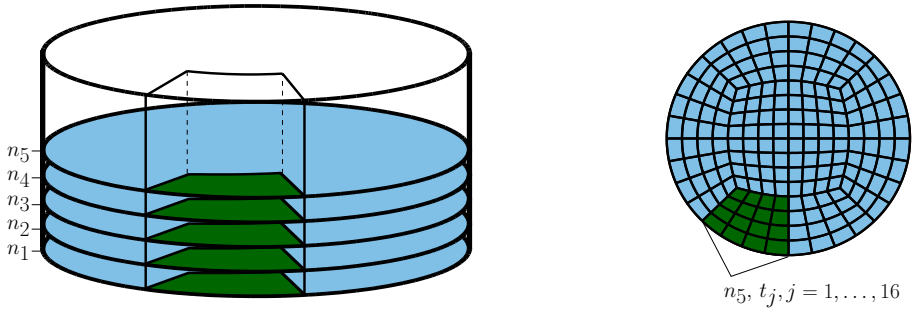
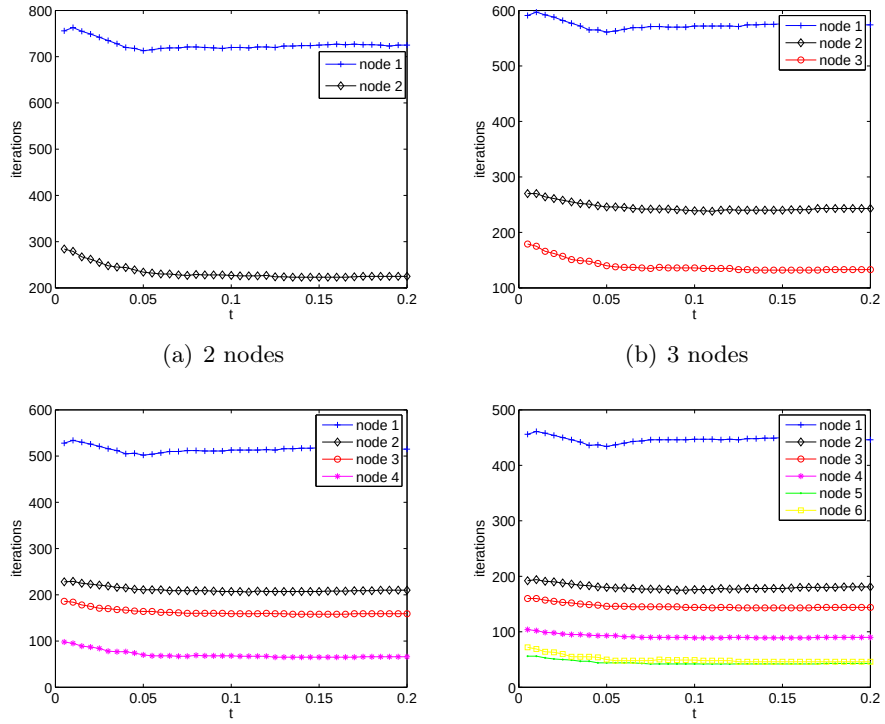


Figure 7: An alternative example of a problem division if we use a combined `MPI`/`OpenMP` approach with data reshuffling. Nodes are handling a number of whole subproblems and threads are used to do operator evaluations in parallel.



(a) 2 nodes

(b) 3 nodes

(c) 4 nodes

(d) 6 nodes

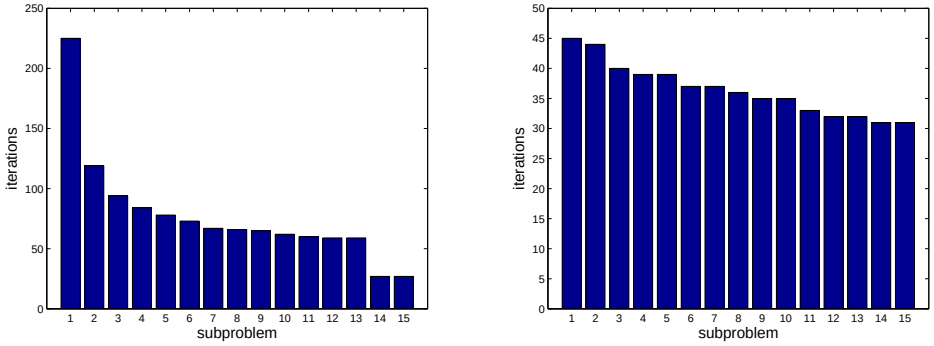
Figure 8: We here integrate a Bénard-Marangoni problem from $t = 0$ to $t = 0.2$ using $\Delta t = 0.005$. The plots show the sum of the iteration counts for the pressure subproblems on the individual nodes for a few selected cases. The distribution strategy used is a naive (approximately) even number of subproblems per node.

Load balancing

If the subproblems are solved by direct methods, we know that each solution process would require the same amount of time, since each subproblem involves exactly the same number of degrees of freedom. However, when used in combination with iterative methods, the proposed parallelization approach is not automatically load-balanced. The number of

iterations needed to solve the individual subproblems will vary (the eigenvalues in Step 2 are different), and thus also the solution time. A naive, even distribution of the subproblems among the nodes may not scale very well. Consider Figure 8 which shows the workload on the individual nodes in terms of iterations. It is quite evident that the workload on the low-numbered nodes are much higher than on the high-numbered ones, in particular the first node always has a much higher load. In order to explain this, we consider Figure 9(a) which shows how the iterations counts for the subproblems are distributed. They are strongly dominated by a few subproblems, and these are associated with the first few eigenvalues (note that the histogram is sorted); typically there is a factor 8-10 between the first subproblem (most iterations) and the last subproblem (least iterations). This explains the poor load balancing. The low rank nodes are handling the most expensive subproblems, while the high rank nodes are left with the less expensive subproblems.

Figure 9(b) shows a sorted distribution of iterations for a Helmholtz solve. While the number of iterations for each solve varies here as well, there is a much more even distribution. This makes it harder to devise a strategy for evening the workload. In fact, it may even be impossible as we are restricted by only being able to hand full subproblems to the nodes. Since the pressure solve seems to give us larger possibilities for evening the workload, and is definitely the one most plagued by the load balancing issue, we limit our attention to the pressure operator for now.



(a) consistent Poisson operator

(b) Helmholtz operator

Figure 9: Plot showing the distribution of iterations for the different subproblems when inverting the a) consistent Poisson operator and b) Helmholtz operator. The results are strongly dominated by a few subproblems in a), while this effect is much less pronounced in b).

To remedy the load balancing problems, we need a distribution strategy which considers the number of iterations in each subproblem, rather than just the number of subproblems. Since we do not have the iteration counts for the current time level available, we resort to the statistics from the previous time level. In general, the statistics are fairly similar between time levels after startup effects have settled. We first sort the subproblems according to the iteration statistics. This sorted list is then used to assign subproblems to each node, in a manner which attempts to keep the number of iterations per node as close to the mean as possible; see Algorithm 3. With this approach, a node tends to get more and more subproblems the higher its rank is. This would be bad for memory bound problems; however, it grants us exactly the properties of the iteration distribution we seek - approximately even iteration counts among the nodes. The obtained results using this strategy are given in Figure 10. The load balance is now much better. The distribution

is close to the optimal distribution for small number of nodes as well as larger number of nodes. In the case with 6 nodes, we see that the number of available subproblems per node is too small for us to be able to even the load.

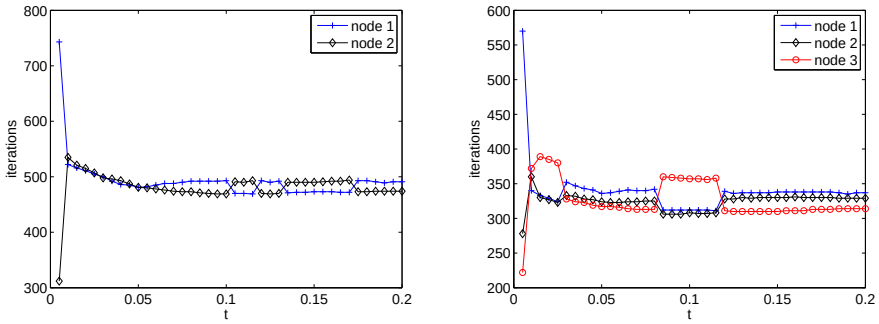
Algorithm 3 Distribute p planes among N nodes in vectors s and c based on iteration statistics in I.

find M, the mean number of iterations per node based on the previous time level's iteration statistics.
sort I, based on iteration count, store order in C.
$front = 0$
$back = p - 1$
for $i = 0$ to $N - 1$ **do**
  while $iters + I(front) < M$ and $front < back$ **do**
    $s(i) = s(i) + 1$
    $c(s(i)) = C(front)$
    $iters = iters + I(front)$
    $front = front + 1$
  end while
  while $iters < M$ and $front < back$ **do**
    $s(i) = s(i) + 1$
    $c(s(i)) = C(back)$
    $iters = iters + I(back)$
    $back = back - 1$
  end while
end for

Algorithm 3 also seems to behave well for the Helmholtz problems. However, we again run into load balancing problems if we use many nodes, as the number of subproblems available per node is too small for us to balance the load through the proposed distribution strategy. For the velocity solves we have one level of parallelism we have not yet exploited, namely that we can solve all three components concurrently. Usually this property is ignored in most parallel codes based on domain decomposition, since each solve should be well load balanced in the first place. Here, however, we can exploit this fact to our benefit. By solving for all three velocity components concurrently, we get approximately three times the number of subproblems. This gives our load balancing algorithm a better chance of evening the workload between the nodes. Figure 11 shows results which proves that we indeed obtain better balancing by solving for all three velocity solves concurrently, in particular for the larger node sizes.

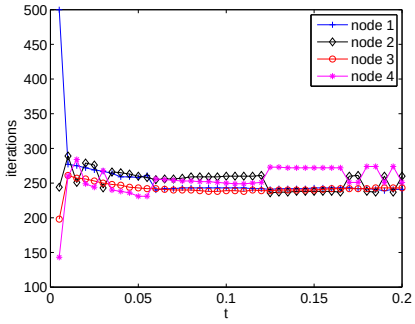
5.3 Operating with the transpose of the eigenmatrix

The final step in the algorithm is another operator evaluation in the x_3-direction (in a data layout sense). Assuming that the data shuffling done prior to solving the subproblems in Step 2 is reversed, this can be performed exactly as in Step 1.

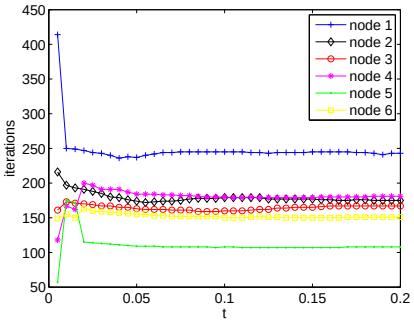


(a) 2 nodes

(b) 3 nodes



(c) 4 nodes



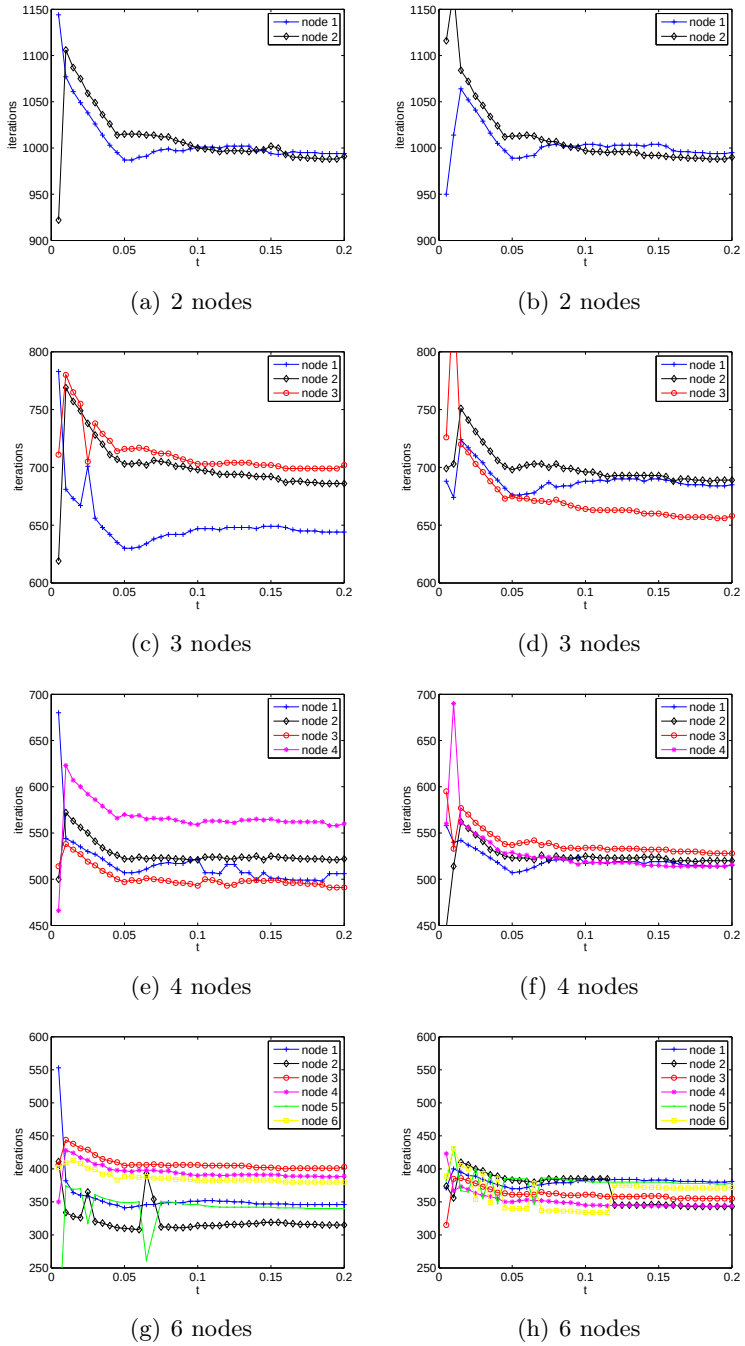
(d) 6 nodes

Figure 10: Sum of the iteration counts as a function of time for the pressure subproblems on the individual nodes for a few selected cases. We here integrate a Bénard-Marangoni problem from $t = 0$ to $t = 0.2$ using $\Delta t = 0.005$. The distribution strategy used is Algorithm 3.

5.4 Why use a combined programming model?

A natural question to ask is why we have chosen to complicate the implementation by using a combined programming model, rather than a more traditional distributed memory model. This is easy to explain in retrospect. The reason is simply that the algorithm maps much better to a combined programming model. The ability to run several solution processes concurrently on each node is imperative for us in order to utilize the parallel nature of the algorithm. If we were using a pure distributed memory model, we would have problems using the proposed parallelization strategies, both with and without data shuffling, due to the fact that each node can only participate in the solution of a single subproblem at any time. If we were to activate as much computing resources as we do using the combined model, we would have to use nodes which would be dorment during the first and third step in the algorithms. This is very bad from an efficiency point of view. One could argue that we could decompose the grid across more nodes in the first step, with each node holding a single super-element in the limit. While the idea is sound, such an approach is not something we would recommend for several reasons. Primarily, it makes intra-node communication a dominant factor during other operations such as direct stiffness summation. This would also make things unnecessarily complicated at the implementation level. Preferably some pattern should be present to ease the organization of the grid operations. The flexibility in the shared memory model, however, is perfect for our needs, since the threads can also easily be utilized during the first and third step of the

algorithm, without any change to the problem partitioning being necessary. Additionally, since each 2D subproblem is solved completely within a single node with our final approach, we avoid the intricacies associated with using `MPI` for fine-grained parallelism, which lessens the implementation effort substantially.



(a) 2 nodes (b) 2 nodes

(c) 3 nodes (d) 3 nodes

(e) 4 nodes (f) 4 nodes

(g) 6 nodes (h) 6 nodes

Figure 11: The left column shows the sum of the iteration counts per node when we perform the velocity solve as three sequential solves, while the right column shows the results when we perform all three velocity solves concurrently. Performing the three solves concurrently substantially improves the load balancing.

6 Speedup results

We first consider results for a Bénard-Marangoni computation using up to 16 processors. This is a coupled thermal-fluid problem, where flow (convection cells) in a fluid is induced by temperature differences. The fluid is governed by the incompressible Navier-Stokes equations, while the temperature is governed by a convection-diffusion equation. We refer the reader to [4] for the details. The calculations are performed in a circular cylinder with a global aspect ratio of

$$\Gamma = \frac{\sqrt{A}}{d} = 8.27,$$

where A is the cross-sectional area and d the height of the container. The cylinder is divided in $K = 48$ spectral elements organized in a single layer ($\mathcal{K} = 48$, $\mathcal{L} = 1$). The polynomial degree within each element is $N = 16$. This means that we have, depending on the boundary conditions, $\mathcal{N}_1 = 15$ or $\mathcal{N}_1 = 16$ subproblems in the velocity and temperature solves and $\mathcal{N}_2 = 15$ in each pressure solve. We integrate this system from $t = 0$ to $t = 0.2$ in steps of $\Delta t = \frac{1}{200}$. At each time level, the solution of this problem consists of four applications of the Helmholtz solver (three for the velocity solves, which are performed concurrently, as well as one for the temperature update, which is performed separately). Additionally, we have one application of the consistent Poisson solver. We also have to integrate a total of 8 scalar convection problems per time level; this is done using explicit time integration [19]. Since the parallelization within the convection problems are based on the domain decomposition as derived while discussing the first step of the elliptic solvers, including these would give better speedup results. We have excluded them from the timing results, since we are mainly interested in the properties of the elliptic solvers (Algorithms 1-2). The speedup results are given in Table 1.

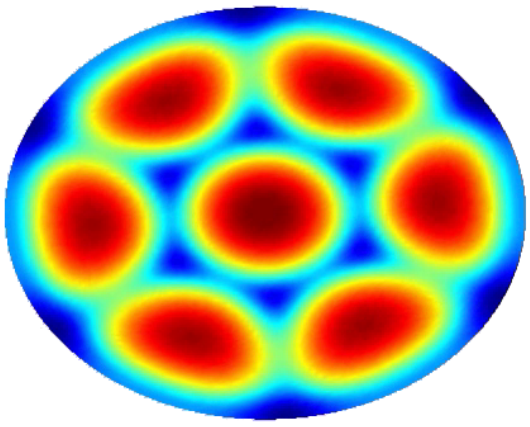


Figure 12: The resulting temperature field on the top of the domain in a Bénard-Marangoni simulation. The container is a circular cylinder with global aspect ratio $\Gamma = 8.27$, while the $Ma = 105, Ra = 48, Pr = 890$. The flow pattern consists of 7 cells.

Table 1: CPU time τ for solving a Bénard-Marangoni problem tabulated as a function of q, the number of `MPI` nodes used. We use $K = 48$ elements organized in a single layer ($\mathcal{K} = 48, \mathcal{L} = 1$), where each element is of order $N = 16$. The system is integrated from $t = 0$ to $t = 0.2$ using time steps of $\Delta t = 0.005$. The first ten time steps are left out of the timings to allow the iteration statistics we base the distribution of problems on to settle. The computations are performed on a IBM Power5 PPC computer.

q	τ_q [s]	$\frac{\tau_1}{\tau_q}$	η_q	$\tau_{q,4}$ (4 threads)	$\frac{\tau_{q,4}}{\tau_q}$	$\eta_{q,4}$
1	762	1.00	-	197	3.9	0.98
2	392	1.94	0.97	123	3.2	0.77
3	264	2.88	0.96	106	2.5	0.60
4	203	3.75	0.94	99	2.1	0.50
6	142	5.4	0.90	-	-	
12	105	7.3	0.61	-	-	

There are two things which are evident from these results. Our efforts to distribute the subproblems evenly among the nodes based on iteration counts is quite successful. Even with 4 nodes, the achieved speedup is still within 6% of being perfect. However, the speedup from utilizing more threads is diminishing for the larger node sizes. This had to be expected, since the number of subproblems per node rapidly approach the number of threads per node. With only a single subproblem per thread, we have no freedom in how these solves are performed. Hence the obtained speedup reflects iteration counts, in the sense that the solution time on each node is dictated by its largest assigned subproblem (in terms of iterations). Unfortunately, this is something we cannot remedy using this parallelization strategy. Following this reasoning, the speedup using 12 nodes is not as bad as it might look. The pressure problems represent the obstacle for increased speedup. We can use iteration counts for the pressure problem as a rough estimate of the maximum achievable speedup. On average, the largest pressure subproblem accounts for about 12% of the total number of iterations used for solving the full pressure problem. This gives an upper bound for the speedup at approximately $1/0.12 \sim 8.333$.

In conclusion, it seems like utilizing more than 8-10 processors for this problem is just a waste of CPU hours. Nonetheless, if we assume we gain a factor 10 from utilizing the tensor-product solvers compared to a standard 3D solver (in a serial context), we would have to utilize 80 processors with a standard domain decomposition method to be on par performance wise. With $K = 48$ elements, this would be challenging since this leaves less than one spectral element per node.

To get an idea how well this scales for larger problem sizes, we also consider the speedup for the two solvers using a second grid. The geometry is again the same circular cylinder, however we now divide it in $K = 480$ elements. These are organized in $\mathcal{L} = 10$ layers of elements in the x_3-direction, with each layer consisting of $\mathcal{K} = 48$ elements. Each spectral element has a polynomial degree of $N = 10$. This means that each 2D subproblem consists of 48 elements, just as in the earlier tests, but that we now have approximately 90 subproblems to divide between the nodes. Since we have more subproblems, we can use more nodes. To get an idea of how well the new algorithms compare to a standard 3D implementation, we also give idealized solution times for a traditional solver. This solver inverts the global 3D operator using an iterative solver with an overlapping Schwarz type preconditioner; see [5, 24] for details. Since the speedup in a real implementation will not be

perfect, using idealized speedup serves as a conservative point of comparison. The results for a Laplace problem are given in Table 2, while the results for the consistent Poisson operator are given in Table 4. The speedup results are, as expected, not perfect here either. However, it seems that even when compared to idealized speedup, we fare very well compared to the 3D reference solver. For instance, for the pressure solve, we would have to utilize approximately 1100 processors in the 3D reference solver to obtain the solution in the same wall clock time as we have here obtained with the tensor-product solver using 48 processors. And we would again run into the problem of having 480 elements to divide among those 1100 processors, making a standard domain decomposition hard to utilize.

Table 2: CPU time τ for the solving a Laplace problem tabulated as a function of q, the number of `MPI` nodes used. We use $K = 480$ elements organized in ten layers ($\mathcal{K} = 48, \mathcal{L} = 10$), where each element has polynomial order $N = 10$. To get more realistic results we solve the same problem 10 times. This for two reasons; primarily we have to do at least two solves to obtain realistic iteration statistics. Secondly, we choose to do 10 solves to smooth out potential data noise. For the 3D reference solver, only the single processor case corresponds to a real calculation. For the other number of nodes ($q > 1$), we give the time we would get if we had perfect speedup.

q	τ [s]	$\frac{\tau_1}{\tau_q}$	η_q	$\tau_{q,4}[s]$ (4 threads)	$\frac{\tau_q}{\tau_{q,4}}$	$\eta_{q,4}$	$\tau_{3d}[s]$	$\frac{\tau_{3d}}{\tau}$
1	59.8	1.0	1.00	14.6	4.0	1.00	502	8.4
3	20.2	2.9	0.97	5.53	3.7	0.90	167	8.3
4	16.4	3.6	0.90	4.20	3.9	0.89	126	7.7
6	10.2	5.9	0.98	3.21	3.2	0.78	84	8.2
12	5.44	11.0	0.92	1.85	3.0	0.67	42	7.7

Finally we report the speedup obtained in a real simulation. We again consider the same cylinder as earlier, except that we here reduce the order of the elements to $N = 13$. We integrate the system of equations from $t = 0$ to $t = 30$ using time steps of size $\Delta t = \frac{1}{200}$. Both approaches use a total of 12 processors. In the 3D reference solver, these are naturally divided into 12 nodes. For the tensor-product approach, we use 4 nodes, each with 3 threads. This means that there is not much room for load balancing the tensor-product algorithm, in particular since the total number of processing cores equals the total number of subproblems. Hence we would expect a lower gain from using the new algorithm. The temperature field on the top of the domain at the end of the calculation is depicted in Figure 12. The speedup results are given in Table 3. Even for this moderately small problem, we have a speedup close to 6 compared to a traditional solver and a traditional domain decomposition parallelization.

Table 3: CPU time and speedup for a full Bénard-Marangoni simulation. The container is a circular cylinder with global aspect ratio $\Gamma = 8.27$. The nondimensional numbers used here are Marangoni number $Ma = 105$, Rayleigh number $Ra = 48$, Prandtl number $Pr = 890$.

3D reference solver	tensor-product	speedup
13h42m	2h22m	5.8

Table 4: CPU time τ for the doing the pressure update in a Bénard-Marangoni problem tabulated as a function of q, the number of `MPI` nodes used. We use $K = 480$ elements organized in ten layers ($\mathcal{K} = 48, \mathcal{L} = 10$), where each element has polynomial order $N = 10$. To get more realistic results we solve the same problem 10 times. This for two reasons; firstly we have to do at least two solves to obtain realistic iteration statistics. Secondly, we choose to do 10 solves to smooth out potential data noise. For the 3D solve, only the single processor corresponds to a real calculation. For the other number of nodes ($q > 1$), we give the time we would get if we had perfect speedup.

q	τ [s]	$\frac{\tau_1}{\tau_q}$	η_q	$\tau_{q,4}[s]$ (4 threads)	$\frac{\tau_q}{\tau_{q,4}}$	$\eta_{q,4}$	$\tau_{3d}[s]$	$\frac{\tau_{3d}}{\tau}$
1	102	1.0	1.00	24.9	4.0	1.00	3514	34
3	35.5	2.9	0.97	9.40	3.8	0.90	1171	33
4	30.1	3.4	0.85	9.20	3.3	0.69	879	29
6	19.5	5.2	0.87	5.64	3.5	0.75	586	30
12	10.4	9.8	0.82	3.17	3.3	0.67	292	28

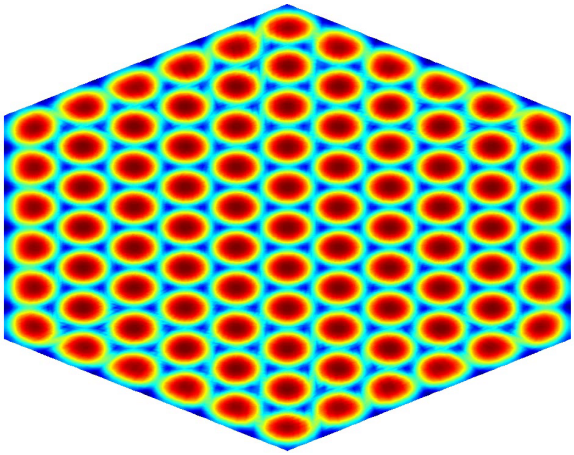


Figure 13: The resulting temperature field on the top of the domain in a Bénard-Marangoni simulation. The container is a hexagonal cylinder with a global aspect ratio $\Gamma = 30.0$. The nondimensional numbers used here are Marangoni number $Ma = 105$, Rayleigh number $Ra = 48$, Prandtl number $Pr = 890$. The flow pattern consists of 91 cells.

7 Summary and conclusions

We have designed and implemented a parallel realization of the tensor-product solvers for partially deformed geometries suitable for fluid flow problems described in [5, 29]. The new code has been compared to a standard 3D approach under idealized conditions and been shown to solve these problems using substantially less computational resources. In fact, the results show that even if we consider computing resources cheap, traditional tools

would still not be able to extract the same performance in terms of wall clock time. That is, even if we use more processors for the same problem, this cannot be expected to alleviate the gains obtained by using the new algorithm instead of more traditional tools.

In this work we have considered applications using up to 48 processors. Today this is considered to be relatively few processors, since modern supercomputers typically consists of thousands of processing cores. However, this is not due to a limitation of the proposed scheme, but rather just reflects the particular applications we have had in mind. The suggested parallelization scheme should scale well to many more processors, as long as the problem size grows accordingly. Even if the problem considered is of a size that requires the employment of parallel matrix-matrix multiplications in the pre- and post-transform steps (Step 1 and Step 3), this would not require a major change in strategy. The proposed scheme can simply be applied within each "layer" of nodes, and thus the changes to the implementation would be minimal.

In [4] we present Bénard-Marangoni results obtained using the new solver, including simulations of larger systems than what have been reported in the literature earlier. An example of such a simulation is given in Figure 13. The system was integrated for approximately 72h (3 days) using 12 processors (4 nodes, each with 3 threads). Assuming we gained a factor 10 in speedup from using the tensor product algorithm, obtaining the result would have taken approximately 720h, or 30 days, using the same amount of computing resources with traditional tools. This serves as an example that these algorithms can be extremely useful when they can be utilized.

8 Acknowledgement

The work has been supported by the Norwegian University of Science and Technology and the Research Council of Norway under contract 159553/I30. The support is gratefully acknowledged.

References

[1] The MPI specification. http://www.mpi-forum.org.

[2] The OpenMP specification. http://openmp.org.

[3] K. Arrow, L. Hurwicz, and H. Uzawa. *Studies in Nonlinear programming*. Standford University Press, 1958.

[4] H. Bénard. Les tourbillons cellulaires dans une nappe liquide transportant de la chaleur par convection en régime permanent. *Annales de Chimie et de Physique*, 23:62–144, 1901.

[5] T. Bjøntegaard, Y. Maday, and E. M. Rønquist. Fast tensor-product solvers: Partially deformed three-dimensional domains. *J. Sci. Comput.*, 39:28–48, 2009.

[6] T. Bjøntegaard and E.M. Rønquist. Simulation of three-dimensional Bénard-Marangoni flows including deformed surfaces. *Communications in Computational Physics*, 5(2–4):273–295, 2009.

[7] M. J. Block. Surface tension as the cause of Bénard cells and surface deformation in a liquid film. *Nature*, 178:650–651, 1956.

[8] M.S. Carvalho and L.E. Scriven. Three-dimensional stability analysis of free surface flows: application to forward deformable roll coating. *J. Comput. Phys.*, 151:534–562, 1999.

[9] A. J. Chorin. Numerical solution of the Navier-Stokes equations. *Math. Comput*, 22, 1968.

[10] D. Chu, R. Henderson, and G. E. Karniadakis. Parallel spectral-element-Fourier simulation of turbulent flow over riblet-mounted surfaces. *Theoretical and Computational Fluid Dynamics*, 3:65–112, 1992.

[11] D. Chu, R. Henderson, and G. E. Karniadakis. Parallel spectral-element-Fourier simulation of turbulent flow over riblet-mounted surfaces. *Theoretical and Computational Fluid Dynamics*, 3(4):219–229, 1992.

[12] M. Dryja and O. B. Widlund. Towards a unified theory of domain decomposition algorithms for elliptic problems. In Tony Chan, Roland Glowinski, Jacques Périaux, and Olof Widlund, editors, *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 3–21. SIAM, Philadelphia, PA, 1990.

[13] P.F. Fischer. An overlapping Schwarz method for spectral element solution of the incompressible Navier-Stokes equations. *J. Comput. Phys.*, 133:84–101, 1997.

[14] K. Goda. A multistep technique with implicit difference schemes for calculating two- or three-dimensional cavity flows. *J. Comput. Phys.*, 30, 1979.

[15] R. Henderson. Nonlinear dynamics and pattern formation in turbulent wake transition. *J. Fluid Mech.*, 353:65–112, 1997.

[16] G. E. Karniadakis, M. Israeli, and S. A. Orszag. High-Order Splitting Methods for the Incompressible Navier-Stokes Equations. *J. Comput. Phys.*, 97:414–443, 1991.

[17] E. L. Koschmieder. *Bénard Cells and Taylor Vortices*. Cambridge University Press, 1993.

[18] E. L. Koschmieder and M. I. Biggerstaff. Onset of surface–tension–driven Bénard convection. *J. Fluid Mech.*, 167:49–64, 1986.

[19] A. M. Kvaring, T. Bjøntegaard, and E. M. Rønquist. On pattern selection in three-dimensional Bénard-Marangoni flows. *Submitted to Communications in Computational Physics*, 2010.

[20] A. M. Kvarving, T. Bjøntegaard, and E. M. Rønquist. A fast tensor-product solver for incompressible fluid flow in partially deformed three-dimensional domains. *In preparation*, 2010.

[21] R. E. Lynch, J. R. Rice, and D. H. Thomas. Direct solution of partial difference equations by tensor product methods. *Numer. Math.*, 6:185–199, 1964.

[22] Y. Maday, D. Meiron, A. T. Patera, and E. M. Rønquist. Analysis of iterative methods for the steady and unsteady Stokes problem: Application to spectral element discretizations. *J. Sci. Comput.*, 14:310–337, 1993.

[23] Y. Maday and A. T. Patera. Spectral element methods for the incompressible Navier-Stokes equations. In *State-of-the-art surveys on computational mechanics (A90-47176 21-64). New York, American Society of Mechanical Engineers*, pages 71–143, 1989.

[24] Y. Maday, A. T. Patera, and E. M. Rønquist. An Operator-Integration-Factor Splitting Method for Time-Dependent Problems: Application to Incompressible Fluid Flow. *J. Sci. Comput.*, 5(4), 1990.

[25] Y. Maday, A. T. Patera, and E. M. Rønquist. The $P_N \times P_{N-2}$ method for the approximation of the Stokes problem. Technical Report 92009, Department of Mechanical Engineering, Massachusetts Institute of Technology, 1992.

[26] M. Medale and P. Cerisier. Numerical simulation of Bénard-Marangoni convection in small aspect ratio containers. *Numerical Heat Transfer, Part A*, 42:55–72, 2002.

[27] S. A. Orszag, M. Israeli, and M. O. Deville. Boundary Conditions for Incompressible Flows. *Journal of Scientific Computing*, 1(1):75–111, 1986.

[28] R. Temam. Sur l'approximation de la solution des équations de Navier-Stokes par la méthode des par fractionnaires ii. *Arch. Ration. Metch. Anal*, 33:377–385, 1969.

[29] A. Thess and S.A. Orszag. Surface-tension-driven Bénard convection at infinite Prandtl number. *J. Fluid Mech.*, 283:201–230, 1995.

[30] A. Toselli and O. B. Widlund. *Domain Decomposition Methods - Algorithms and Theory*, volume 34 of *Springer Series in Computational Mathematics*. Springer, 2004.

[31] J. van Kan. A second-order accurate pressure-correction scheme for viscous incompressible flow. *J. Sci. Stat. Comput.*, 3, 1986.