Boundary Integral Equations - Project
Nearly Singular Integrals and Special Quadrature

Fredrik Fryklund and Sara Pålsson

January 20, 2017

Introduction

This project deals with the optimization and parallelization of solving linear equations with boundary integral equation (BIE) methods. To demonstrate the strengths and weaknesses with a BIE method, and how to make the computations for these faster, we use the example problem of solving Laplace's equation in two-dimensions on a starfish domain. Therefore, before going into details on our existing code, performance analysis and parallelization, we start by overviewing the methods we will use.

Boundary Integral Equations

Boundary integral equations (BIE) provide a way to solve linear equations to a high accuracy. By reformulating the differential equation as an integral equation over the surface of our domain, we reduce the degrees of freedom of the problem compared to a standard grid-based method where we discretize the whole domain, for example finite element and finite difference methods. A simple sketch of the differences between a standard grid-based method and a BIE method can be seen in Fig. 1, where we have used squared elements for our standard grid and discretized the BIE with equidistant points along the boundary. Here, show the discretization for a starfish domain in two dimensions, which is smooth. This is going to be our example domain throughout this report.

For completeness, we state Laplaces equation in our domain $\Omega \in \mathbb{C}$,

$$\begin{aligned} \Delta u &= 0, \ z \in \Omega, \\ u(z) &= f(z), \ z \in \delta\Omega, \end{aligned} \tag{1}$$

with the analytic function $f(z) = \frac{1}{z-z_1} + \frac{1}{z-z_2} + \frac{1}{z-z_3}$, $z_1, z_2, z_3 \notin \Omega$, as boundary condition. As f is analytic, the solution to (1) is known as $u(z) = f(z)$ for all $z \in \Omega$. This provides us with an easy way to check the errors from our different quadrature schemes. For a boundary integral method, solving (1) means to first compute the complex density $\mu(z)$ for all $z \in \delta\Omega$ through

$$\frac{1}{2}\mu(z) + \frac{1}{2\pi}\int_{\delta\Omega} \mu(\tau)\Im\left\{\frac{d\tau}{\tau - z}\right\} = f(z). \tag{2}$$

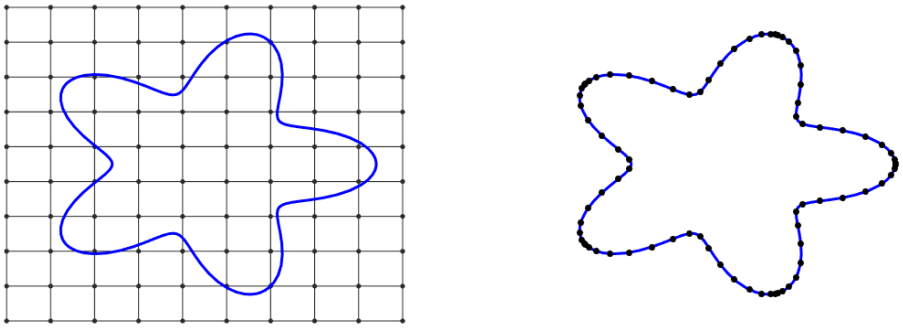

Figure 1: Typical examples of grid-based discretization (left) and BIE discretization (right) of a starfish domain.

For the points $z = \tau$ on the boundary, we have the limit

$$\lim_{z\to\tau} \Im\left\{\frac{\tau'}{\tau - z}\right\}) = \Im\left\{\frac{\tau''}{\tau - z}\right\}.$$

When we have computed μ, we can compute the solution $u(z)$ at all points z both on the domain and in the boundary, through

$$u(z) = \frac{1}{2\pi}\int_{\delta\Omega} \mu(\tau)\Im\left\{\frac{d\tau}{\tau - z}\right\}. \tag{3}$$

There are several advantages with a boundary discretization: the reduction of degrees of freedom means we will have a smaller system to solve in (2), we know the position of the interface explicitly and do not need to interpolate from a domain-based grid, and we can easily track movement of the interface. Other methods for tracking interfaces, such as immersed boundary methods, perform only to low orders of accuracy, whilst we with a BIE obtain spectral accuracy. The accuracy of a BIE method is decided by the quadrature, and if we use a high-order quadrature method to evaluate our integrals our method will be of high order. There are of course also drawbacks with this kind of methods. Firstly, the system we need to solve to compute μ in (2) is dense (albeit smaller than that of a grid-based scheme due to the decrease in degrees of freedom). However, algorithms such as *gmres* still converge quickly, as our condition numbers are more or less constant and also small.

As mentioned previosuly, the integrals we evaluate will contain singularities. These do not pose any problem, as we can use known limits or singularity subtraction to handle such cases. However, as we evaluate our integral in (3) close to the boundary of our domain, the integral will become so-called "nearly singular". This means that whilst the integrand mathematically is smooth and well-defined, numerically it will be difficult to resolve and our errors will increase dramatically. To see this specifically, we look at the integrand

$$k(\tau, z) = \Im\left\{\frac{\tau'}{\tau - z}\right\}. \tag{4}$$

As z becomes closer and closer to τ, this will give us a denominator that is approaching zero, and the integrand will therefore grow rapidly. This will be increasingly difficult to resolve, and simply upsampling the discretization on the boundary $\delta\Omega$ will not be sufficient. For these cases, we need

to do something different and will apply a special quadrature.

A short summation of the special quadrature follows. For a more detailed description, see [1]. When it comes to solving (3) with special quadrature, we use the fact that the kernel (4) is the same as in the monomials

$$p_k(z) = \int \frac{\tau^k}{\tau - z} d\tau, \; k = 0, \ldots, 31. \tag{5}$$

For $p_0(z)$ we have the analytical solution

$$p_0(z) = \log(1 - z) - \log(-1 - z). \tag{6}$$

The other p_k can be obtained through a simple recursion. Thus, for a panel P_j we can write

$$\int_{P_j} \mu(\tau) \Im \left\{ \frac{d\tau}{\tau - z} \right\} = \Im \left\{ \sum_{k=0}^{31} c_k \int_{P_j} \tau^k \frac{d\tau}{\tau - z} \right\} = \Im \left\{ \sum_{k=0}^{31} c_k p_k \right\}, \tag{7}$$

where $\mu_j(z) = \sum_{k=0}^{31} c_k \tau^k$ is the polynomial interpolation of μ restricted to panel P_j.

As a demonstration for how the special quadrature works, we show in Fig. 2 how the error in u behaves as we approach the boundary $\delta\Omega$. The number of panels influences the error, but as can be seen upsampling does not completely remove the problems. In comparison, in Fig. 3 we see the error after applying the special quadrature. Also for 35 panels, the errors are negligble.

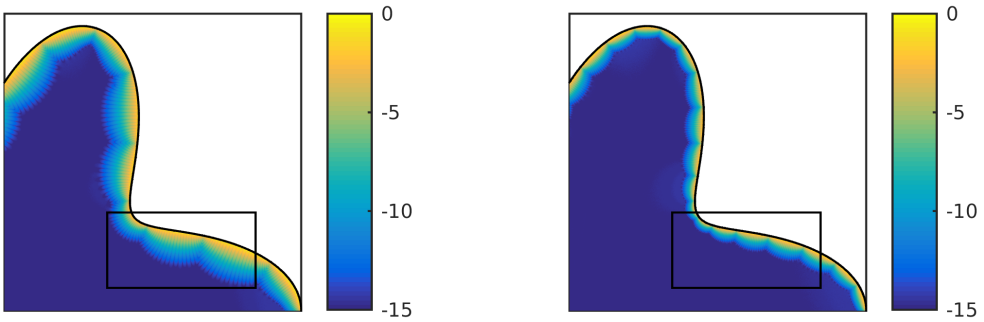

Figure 2: Example of error in domain when solving Laplace's equation, using only normal quadrature, using 35 panels (left) and 70 panels (right).

If we zoom in and compare the level curves for the error in the black box, we see that the special quadrature removes all errors up to $\approx 10^{-15}$, see Fig. 4.

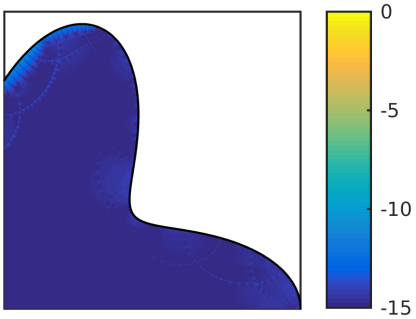

Figure 3: Error after application of special quadrature, for 35 panels.

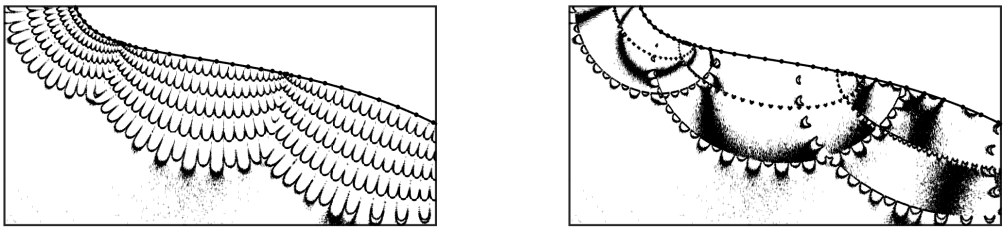

Figure 4: Error curves, for normal quadrature, $\log_{10} e(z) = \{-15, -12, -9, -6, -3\}$ (left) and special quadrature (only $\log_{10} e(z) = -15$ visible) (right).

The Algorithm

The algorithm to compute the boundary integral solution is divided up into several parts. The three major ones are to compute the complex density μ through (2), compute the solution u everywhere in the domain through (3) and finally to compute corrections to u using special quadrature in those cases it is needed. When discretizing our boundary $\delta\Omega$, we divide it into panels on which we put 16 Gauss-Legendre nodes each. The domain is discretized radially and angularly, going from the middle outwords towards $\delta\Omega$ anti-clockwise. Here, we describe each algorithm briefly. The algorithms can be found in Appendix A.

First, in Alg. 1 we solve (2) for the complex density μ. This is done by first filling the system matrix A, through two *for*-loops. We then compute μ through `gmres`. Our matrix A is dense and real-valued, but not symmetric. The major parts of the algorithm are thus two for-loops of the **boundary** discretization points, and a call to *gmres*.

The next step is to compute the solution in the domain through (3). We do this by integrating our μ over the boundary $\delta\Omega$, using the same discretization as in Alg. 1. This becomes one for-loop over the **domain** discretization points and one over the **boundary** discretization points (typically fewer than those on the boundary), see Alg. 2.

The final major step of the algorithm is to apply the special quadrature [1] to reduce the errors in u close to the boundary. This algorithm is briefly described in Alg. 3. It consists of a for-loop over all domain points τ_i, and then a for-loop over all panels of the domain discretization P_j. For each panel and point pair, the distance between them is computed. If the distance is less than a set threshold, further investigations are needed. We will compute an analytical and numerical value of p_0 (6), which contains the same kernel as our solution integral. If these two values of p_0 differ by less than a constant, our integral is deemed well resolved. If not, we upsample to 32 points Gauss-Legendre quadrature on the panel, and compare p_0 again. Now, if the upsampled integral gives a good value of p_0 we assume that also our integral for u_i will be well resolved for point τ_i on panel P_j. We therefore compute a new value for u_i on P_j only, and add it to u_i. If upsampling is not enough, we apply our special quadrature. We then need to compute all p_k through a recursion, (5), and then find the coefficient to interpolate this to a polynomial of degree 15. This is done through solving a Vandermonde system. We then compute our new value of u_i on panel P_j with the specal quadrature (7).

Performance of Serial Code

We analyze our code using CrayPAT as well as a function to time the separate functions individually without CrayPAT (convenient when making several runs). We test both the dependency on the number of panels N_{pan} as well as the number of discretization points N_{dom}.

Changing Domain Discretization

Firstly, we investigate how our code changes as we change the number of domain points, N_{dom}. In these tests, we have set $N_{pan} = 20$ which gives us an error of $\approx 10^{-8}$. For some different N_{dom} ranging from 25 to 13 million we see the timings in Fig. 5. For small N_{dom}, the total computation time is dominated by the time it takes to solve the linear system when computing the complex density μ. For this example that cost will be constant as we keep N_{pan} constant, and μ is independent of the domain discretization points. As N_{dom} grows, the cost will be dominated by the computation of u with standard quadrature and the special quadrature. They both show the same cost, which is natural as the main feature of both algorithms is a for-loop through all points. We can also see this as they show complexity $O(N)$, which a for-loop would.

Performance Analysis

Next, we investigate the code with CrayPAT as we change N_{dom}. We start by studying a problem were we are still using a relative small amount of points in our domain, i.e. were *solveDensity*, Alg. 1, dominates the cost. This problem is run for $N_{dom} = 1600$, $N_{pan} = 20$. When studying the performance through CrayPAT, we see that the call to *gmres* takes most time, see Fig. 6. We set that the functions being called the most times, such as *IPMultR* and *tau* e.g., are very quick,

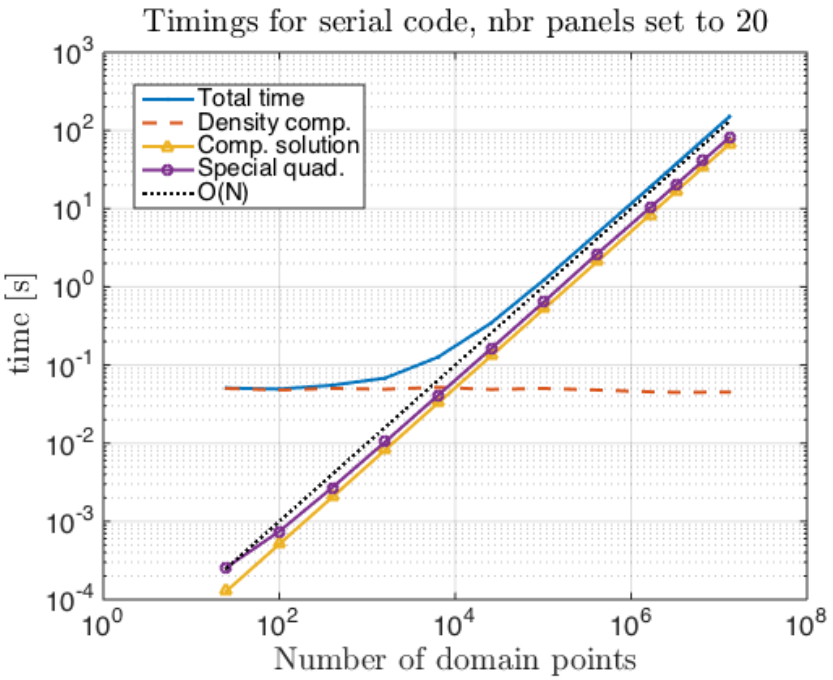

Figure 5: Timings serial code for varying N_{dom}. Fixed $N_{pan} = 20$.

whereas the three major algorithms pin-pointed earlier indeed take the most time. We see in the right pie chart, that it is the *gmres*-call that takes up the most time. Furthermore, if we investigate (USER) cache misses, we see that our code has a quite bad hit/miss ratio, see Tab. 1. Our total D1 cache hit/miss ratio, however, seem to be decreased by the function call to *call_gsl_gmres*, which is the *gmres* solver in the GSL-package. Other bad results worth noticing and investigating further, seem to be the poor D2 hit/miss ratio by *computeSolution*.

Function	D1 hit/miss (%/%)	D2 hit/miss	D1+D2 hit/miss	Time (%)
Total	47.9/52.1	28.3/71.7	62.7/37.3	100
call_gsl_gmres	23.5/76.5	31.4/68.6	47.5/52.5	70.9
specialquadlapl	97, 8/2.2	100/0	100/0	11.4
computeSolution	100/0	0/100	38.1/61.9	8.9
gl16	98.5/1.5	14.2/85.8	79.82/9.98	2.7
vandernewton	99.8/0.2	100/0	100/0	2.6
solveDensity	99.8/0.2	51.3/48.7	99.9/0.1	2.5

Table 1: D1 and D2 cache hit and miss ratios for serial code, $N_{dom} = 1600$. Data generated by CrayPAT.

Now, we study the timings and cache missed for a problem with larger N_{dom}, i.e. when those computations dominates over the panel computations. We set $N_{dom} = 3276800$ and keep, as before, $N_{pan} = 20$. In Fig. 7 (left) we see that the number of function calls is dominated, still, by *IPmultR* and *vandernewton*. The functions *tau, taup, taupp* no longer influence the computations as they are called only once in the beginning of the simulation, to set up the boundary domain. Regarding the time consumption, it is now clear (right figure) that the call to *gmres* is no longer of

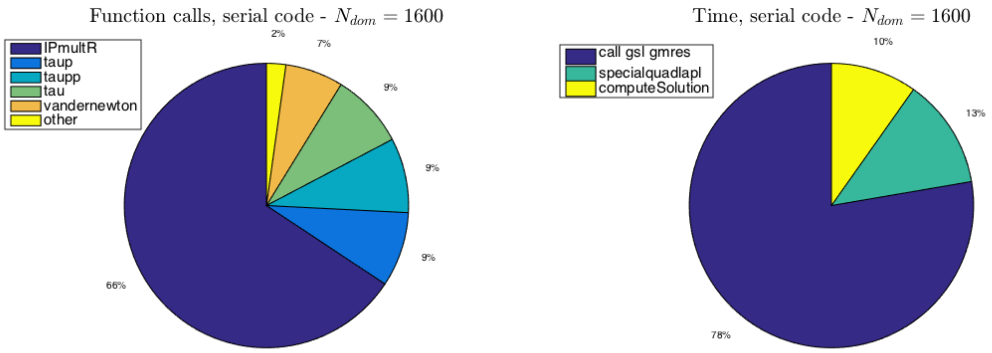

Figure 6: Number of function calls and times for serial code $N_{dom} = 1600$, $N_{pan} = 20$.

any significance. Now the computation is dominated instead by *specquadlapl* and *computeSolution*.

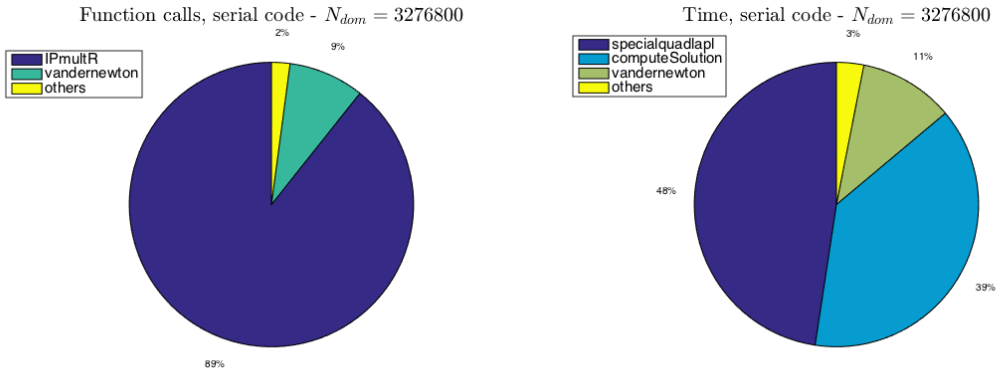

Figure 7: Number of function calls and times for serial code $N_{dom} = 1600$, $N_{pan} = 20$.

In comparison to the example with smaller N_{dom}, our cache utilization has now significantly approved among the costly functions. For all functions which make up the majority of the execution time all have D1 cache hit and miss ratios of $> 99\%$. Also the D1+D2 cache hit,miss ratios are all $> 99\%$. However, the D2 cache hit,miss ratios are worse, ranging from 55% to 80.1%.

Function	D1 hit/miss (%/%)	D2 hit/miss	D1+D2 hit/miss	Time (%)
Total	99.3/0.7	67.8/32.2	99.8/0.2	100
specialquadlapl	99.1/0.9	72.1/27.9	99.7/0.3	47.6
computeSolution	100/0	80.1/19.9	100/0	38.5
vandernewton	99.4/0.6	55/45	99.7/0.3	10.8
IPmultR	99.3/0.7	68.8/31.2	99.8/0.2	1.6

Table 2: D1 and D2 cache hit and miss ratios for serial code, $N_{dom} = 3276800$. Data generated by CrayPAT.

Speeding Up Our Code

From the tests deducted on our serial code, we conclude that in the simulation cases we are interested in studying, i.e. for problems dominated by N_{dom} over N_{pan}, it is most important to work with *specquadlapl* and *computeSolution* to achieve a speed-up.

Regarding Table 1 and 2, we see that the functions which take the majority of the execution code show good cache hit/miss ratios. In that regard, it is therefore not necessary to change the code for better cache perfomance.

OpenMP

When parallelizing with OpenMP, we focus on *computeSolution* and *specialquadlapl* as they take the most time for large problems. Since both of them are based on for-loops stepping through all the domain points, which are treated independently, it is straightforward to distribute the points among the threads. We create new thread pools inside the two functions, instead of creating a common thread pool during initialization. This adds overhead in the creation, but is very simple to implement. To achieve good load balancing, we initially distribute our domain discretization points radially. This is because points close to the boundary will be significantly more expensive than those far away. Those close to the boundary will trigger *specialquadlapl*. Moreover, to avoid load imbalancing, we divide the for-loops in this function with the *guided* schedule. This means that the threads receiving chunks of "easy" points will finish sooner and obtain further blocks. As a result we have 0% imbalancing for our OpenMP-code.

Strong Scaling

Firstly, we investigate the strong scaling of our algorithm with OpenMP-parallelization. This is done by keeping the number of discretization points in the domain fixed (and quite large) at $15 \cdot 10^6$, and set $N_{panels} = 20$. In Fig. 8, we see how the computation times scale with the number of OpenMP-threads. Except for a very small deviation for a large number of threads, the time scales linearly. This is no surprise due to the embarassingly parallel nature of the problem (however, still a good pedagogical basis on which to learn the concepts of parallel programming).

For completeness, we also study the cache hit/miss ratios of our parallelized problem in Table

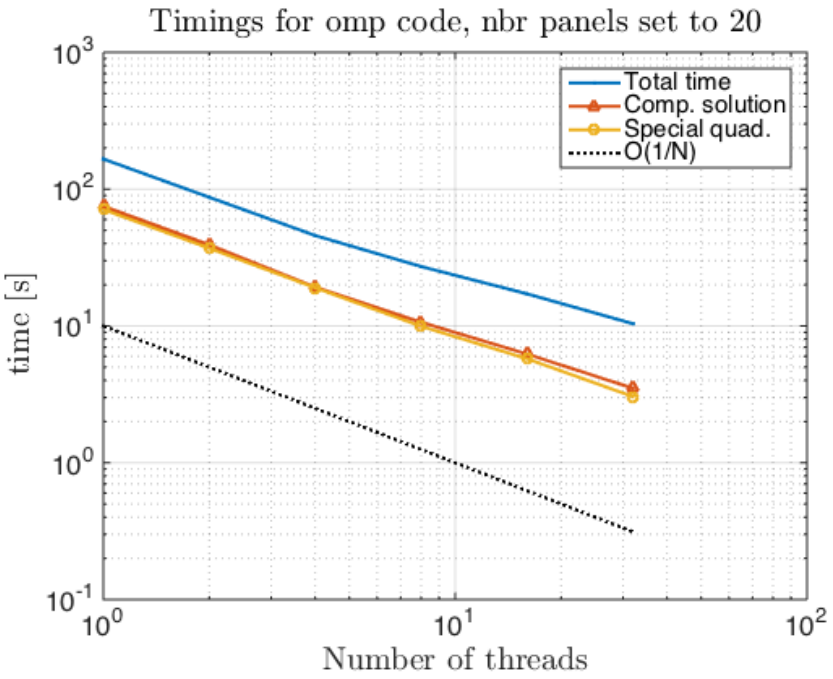

Figure 8: Strong scaling results for OpenMP-parallelized code, $N_{dom} = 15 \cdot 10^6$.

3. As for the larger problem with the serial code, see Table 2, the ratios are good for all our functions. As before, *IPMultR* dominates the number of function calls (85%), but is a small easy function called by *specialquadlapl* and does not dominate the function time (6%), see Fig. 9.

Function	D1 hit/miss (%/%)	D2 hit/miss	D1+D2 hit/miss	Time (%)
Total	98.6/1.4	72.8/27.2	99.6/0.4	100
computeSolution	99.3/0.7	75.0/25.0	99.8/0.2	33.7
specialquadlapl	98.6/1.4	95.5/4.5	99.9/0.1	29.3
create_grid	100.0/0	92.5/7.5	100.0/0	10.4
vandernewton	99.1/0.9	96.2/3.8	100.0/0	6.4
IPmultR	98.5/1.5	93.3/6.7	99.9/0.1	6.0

Table 3: D1 and D2 cache hit and miss ratios for OpenMP-parallelized code, $N_{dom} = 15 \cdot 10^6$. Number of threads 32. Data generated by CrayPAT.

Weak Scaling

To investigate the weak scaling of the problem, we regard a fixed problem size per thread, i.e. $4 \cdot 10^5$ points/thread. When increasing the number of threads, we thus increase also the problem size. Optimally, for a problem like this the computational time should stay constant. As can be seen in Fig. 10, we experience a slight increase in computational time when increasing the number of threads. For the two functions, the increase can probably be explained by the overhead of creating more threads and divide a larger number of discretization points between them. This increase

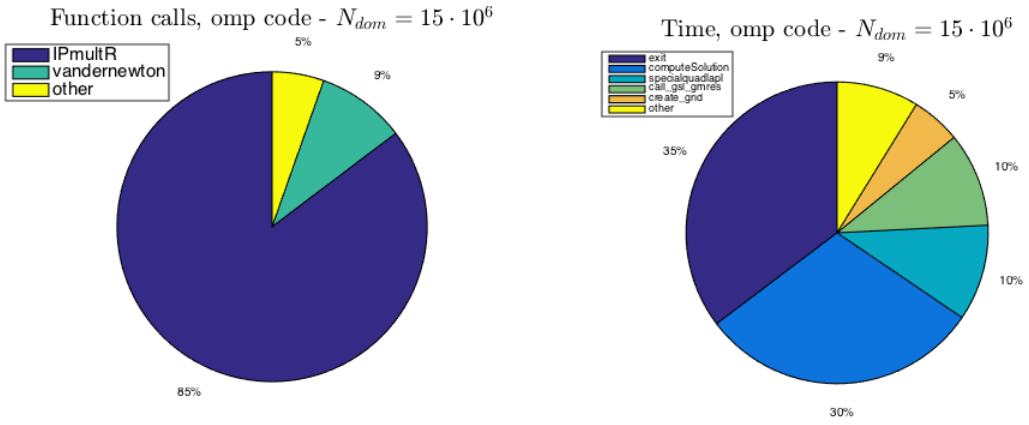

Figure 9: Number of function calls and times for OpenMP-parallelized code $N_{dom} = 15 \cdot 10^6$, $N_{pan} = 20$.

could probably be minimized by working with one pool of threads for both functions. The larger increase in total time comes from the functions which have not been parallelized, e.g. creating the domain.

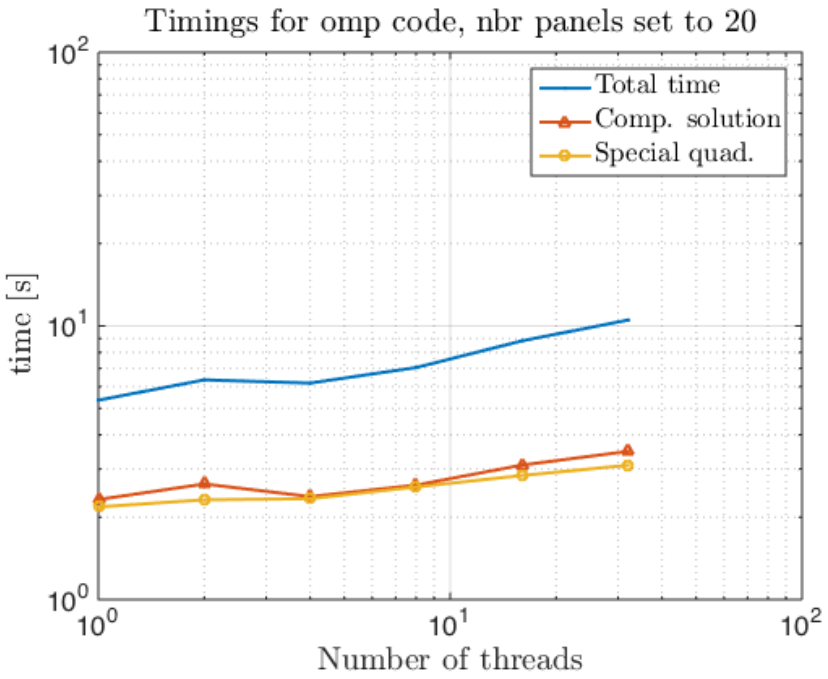

Figure 10: Weak scaling results for OpenMP-parallelized code, $N_{dom}/thread = 4 \cdot 10^5$.

MPI

We parallelize the code with MPI, using a master-slave concept. The master initializes the grid, and computes the density through *solveDensity*. All information needed for *computeSolution*, *specialquadlapl* and *computeError* is then scattered out. When the threads finish their computations the results are then gathered back to the master. Overall this creates a larger overhead than previously with OpenMP, where we could use a suitable scheduling to avoid this problem, as some processes will finish much sooner than others if their point-distribution is favourable. We have tried to initialize our domain discretization in such a way to minimize this problem, but *specialquadlapl* still experiences a load imbalancing of about 70%. In contrast, *computeSolution* has a load imbalance of $\approx 0.01\%$.

Strong Scaling

As for the OpenMP-parallelization, we regard a problem with domain discretization size $15 \cdot 10^6$ and $N_{panels} = 20$. In Fig. 11 we see how the execution time changes when we increase the number of MPI tasks. Note that the actual computation times for both `computeSolution` and `specialquadlapl` decrease linearly with the number of tasks, but the increased computation time for `main function` makes the total time speed-up much less favourable. This is due to load imbalancing, i.e. some tasks finishing slower than others and the master waiting for them. As we can see in Fig. 12, for the strong scaling case, when we increase the number of MPI tasks, the load imbalance time for `specialquadlapl` decreases. This means that the more tasks we have, for a constant number of domain points, we spread the worst-case-scenario points between more threads.

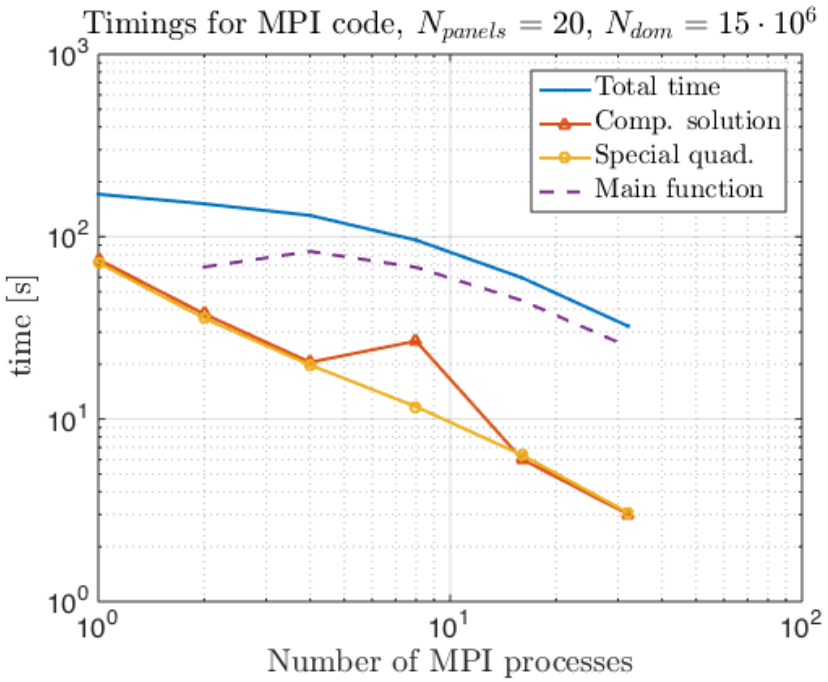

Figure 11: Strong scaling results for MPI-parallelized code, $N_{dom} = 15 \cdot 10^6$.

For completeness, we again plot the cache miss/hit ratios in Table 4, which we still show very good hit/ratio percentages. As previously, the function calls and time is dominated by `IPmultR`, see Fig. 13.

Function	D1 hit/miss (%/%)	D2 hit/miss	D1+D2 hit/miss	Time (%)
Total	99.1/0.9	70.7/29.3	99.7/0.3	100
`main function`	100/0	75.0/25.0	100.0/0	77.3
`specialquadlapl`	99.2/0.8	71.9/28.1	99.8/0.2	9.5
`computeSolution`	100/0	98.5/1.5	100/0	9.3
`vandernewton`	99/1.0	70.2/29.8	99.7/0.3	2.1
`IPmultR`	99.1/0.9	69.4/30.6	99.7/0.3	1.5

Table 4: D1 and D2 cache hit and miss ratios for MPI-parallelized code, $N_{dom} = 15 \cdot 10^6$. Number of MPI-processes 32. Data generated by CrayPAT.

Weak Scaling

When investigating the weak scaling of the problem, Fig. 14, we see that `computeSolution` scales well and remains almost constant. We also see that `specialquadlapl` scales well. However, as the number of tasks increases our `main function` gets more expensive, and thus also the total time. This is because for more processes the amount of communication needed increases, and also the waiting time. This results in the whole algorithm scaling badly in the weak sense. As previously, we have used $4 \cdot 10^5$ points per MPI-task, and $N_{panels} = 20$.

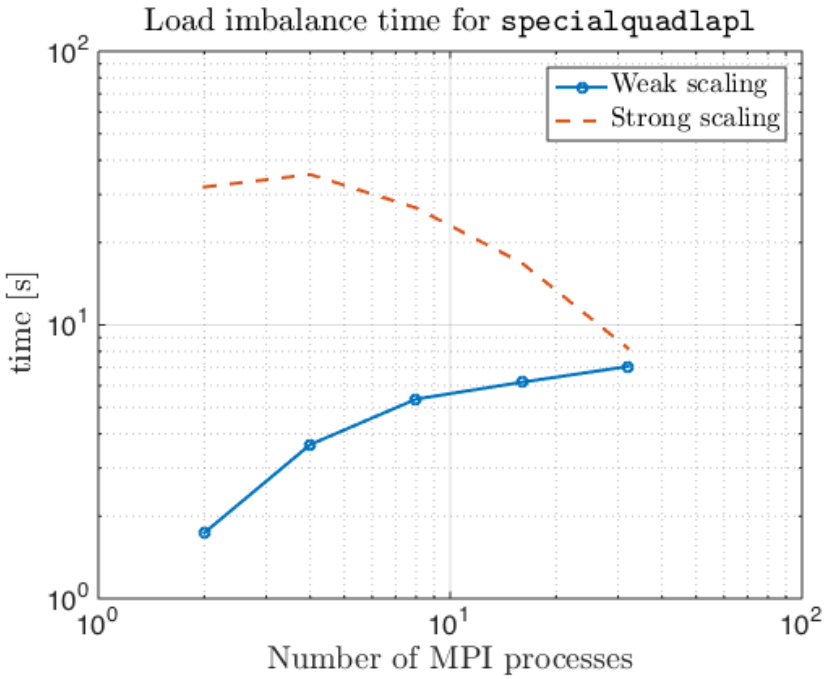

Figure 12: Load imbalance time for `specialquadlapl`, for MPI-parallelized code, strong and weak scaling.

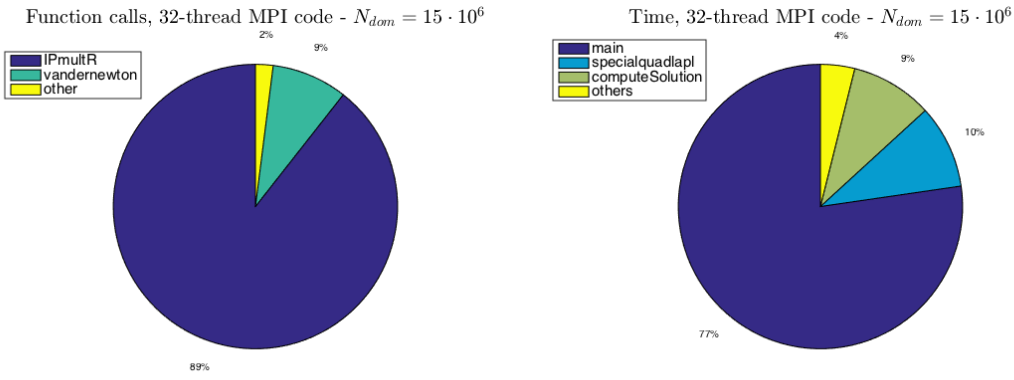

Figure 13: Number of function calls and times for MPI-parallelized code $N_{dom} = 15{\cdot}10^6$, $N_{pan} = 20$.

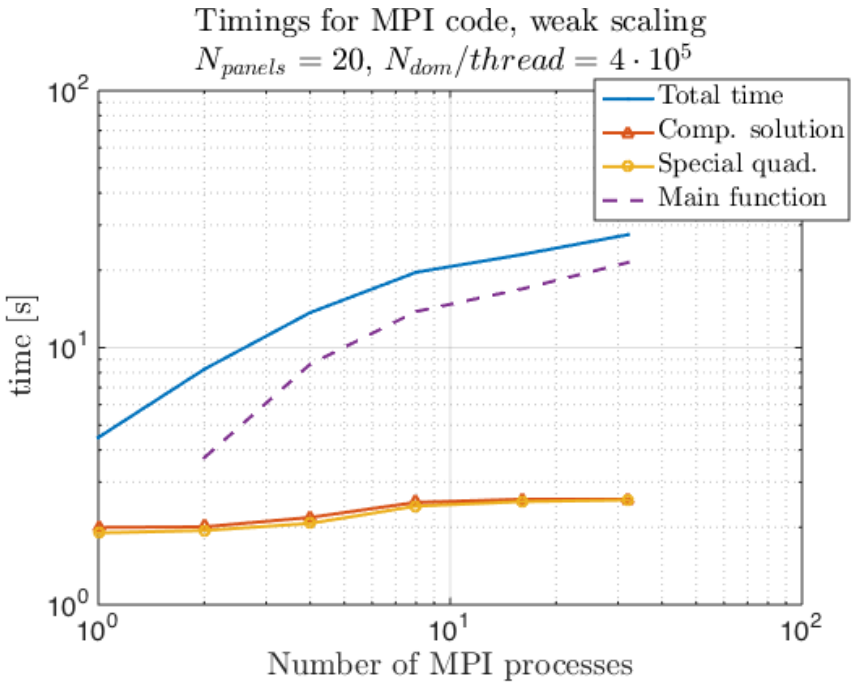

Figure 14: Weak scaling results for MPI-parallelized code, $N_{dom}/thread = 4 \cdot 10^5$.

Hybrid Method

How do we use both OpenMp and MPI?

Plot times

Conclusions

WHat is good, what is worth doing?

References

[1] Helsing, J. and Ojala, R., 2008. On the evaluation of layer potentials close to their sources. *Journal of Computational Physics*, 227(5), pp.2899-2921.

A Algorithms

Algorithm 1: Computing complex density $\mu(z)$, for all $z \in \delta\Omega$.

Function solveDensity (z, zp, zpp, W, RHS);
Input : Interface discretization points and derivatives: z, zp, zpp. Quadrature weights: W. Right hand side: RHS.
Output: μ
for (*go through all boundary points* i)
{
  for (*go through all boundary points* j)
  {
    Fill dense matrix A:
    Elements $A_{ij} = \frac{1}{2\pi} W_j \Im\left\{\frac{zp_j}{z_j - z_i}\right\}$.
    Elements $A_{ii} = \frac{1}{2} + \frac{1}{2\pi} W_i \Im\left\{\frac{zpp_i}{2zp_i}\right\}$.
  }
}
Solve $A\mu = RHS$ with gmres.

Algorithm 2: Computing solution to Laplace's equation u, for all $\tau \in \Omega$.

Function computeSolution (z, zp, τ, W, μ);
Input : Interface discretization points and derivatives: z, zp. Domain discretization points τ. Quadrature weights: W. Complex density μ.
Output: u
for (*go through all domain points* τ_i)
{
  $u_i = 0$;
  for (*go through all boundary points* z_j)
  {
    Compute velocity u_i: $u_i = u_i + W_j \mu_j \Im\left\{\frac{zp_j}{z_j - \tau_i}\right\}$;
  }
}

Algorithm 3: Computing solution to Laplace's equation u with special quadrature, for all $\tau \in \Omega$.

```
1  Function specialQuadrature (τ, W, P_j, μ, u);
   Input  : Interface discretization points and derivatives: z, zp. Domain discretization points
            τ. Quadrature weights: W. Complex density μ. Solution u computed with
            standard quadrature.
   Output: u
2  for (go through all domain points τ_i)
3  {
4      for (go through all panels P_j)
5      {
6          Compute distance between P_j and τ_i, d_ij if (d_ij < TOL)
7          {
8              Investigate this point-panel pair further:
                   • Compute analytical solution to p_0.
                   • Go through all panel points (16) and compute numerical approximation to p_0, p̂_0.
               if ( |p_0 − p̂_0| > TOL)
               {
                   Normal quadrature not sufficient to resolve this domain point. Upsample to
                   32-point Gauss-Legendre panel on P_j, z̃_i, and comupte p̂_0 again. if (
                   |p_0 − p̂_0| > TOL)
                   {
                       Upsample sufficient for resolution. Compute panel contribution anew by
                       going throuhg points z̃_i. Remove panel contribution from 16-points G.-L.
                       from u and add new contribution from 32-points.
                   }
                   else
                   {
                       Upsampling not sufficient. Apply special quadrature by computing all p_k
                       and interpolating, see REF. Then remove panel contribuiton from
                       16-points G.-L. from u and add new solution.
                   }
               }
               else
               {
                   Normal quadrature is sufficient. No recomputation needed.
               }
9          }
10     }
11 }
```