OpenMP parallelization in the NFFT software library

Toni Volkmer

August 30, 2012

We describe an implementation of a multi-threaded NFFT (nonequispaced fast Fourier transform) software library and present the used parallelization approaches. Besides the NFFT kernel, the NFFT on the two-sphere and the fast summation based on NFFT are also parallelized. Thereby, the parallelization is based on OpenMP and the multi-threaded FFTW library. Furthermore, benchmarks for various cases are performed. The results show that an efficiency higher than 0.50 and up to 0.79 can still be achieved at 12 threads.

1 Overview

The NFFT3 library [3] and its MATLAB interface were parallelized using OpenMP [7].

Both the non-parallel version and the multi-thread OpenMP version of the NFFT3 library provide an identical Application Programming Interface (API). This is realized by using distinct library files for both versions. The non-parallel version of the NFFT3 library can be found in `libnfft3.so` and `libnfft3.a`, the multi-thread OpenMP version in `libnfft3_threads.so` and `libnfft3_threads.a`.

For the MATLAB interface, the user has to specifiy at compile time whether the non-parallel or multi-thread OpenMP version should be built.

The following kernels of the NFFT3 library were parallelized using OpenMP:

- `kernel/nfft`:
  - NDFT (nonequidistant discrete Fourier transform)
  - NDFT$^{\mathsf{H}}$ (adjoint nonequidistant discrete Fourier transform)
  - NFFT (nonequidistant fast Fourier transform)
  - NFFT$^{\mathsf{H}}$ (adjoint nonequidistant fast Fourier transform)
- `kernel/nfsft`:
  - NDSFT (nonequidistant discrete spherical Fourier transform, NDFT on the sphere $\mathbb{S}^2 := \{\boldsymbol{x} \in \mathbb{R}^3 : \|\boldsymbol{x}\|_2 = 1\}$)
  - NDSFT$^{\mathsf{H}}$ (adjoint nonequidistant discrete spherical Fourier transform)
  - NFSFT (nonequidistant fast spherical Fourier transform, NFFT on the sphere $\mathbb{S}^2$)
  - NFSFT$^{\mathsf{H}}$ (adjoint nonequidistant fast spherical Fourier transform)

Furthermore, the following examples/applications, which utilize the OpenMP code, are available:

- `examples/nfft/simple_test_threads` (file `simple_test_threads.c`): simple test program for multi-thread version of `kernel/nfft`
- `examples/nfft/nfft_benchomp` (file `nfft_benchomp.c`): benchmark for multi-thread version of `kernel/nfft` (NFFT and NFFT$^{\mathsf{H}}$ in 1D/2D/3D), which outputs speedup plots (as `pgfplots`) into the LaTeX file `nfft_benchomp_results_plots.tex`, see section 5.1 for example plots
- `examples/nfsft/simple_test_threads` (file `simple_test_threads.c`): simple test program for multi-thread version of `kernel/nfsft`
- `examples/nfsft/nfsft_benchomp` (file `nfsft_benchomp.c`): benchmark for multi-thread version of `kernel/nfsft` (NFSFT and NFSFT$^{\mathsf{H}}$), which outputs speedup plots (as `pgfplots`) into the LaTeX file `nfsft_benchomp_results_plots.tex`, see section 5.2 for example plots
- `applications/fastsum/fastsum_test_threads` (files `fastsum_test.c`, `fastsum.c`): multi-thread version of the NFFT-based fast summation
- `applications/fastsum/fastsum_benchomp` (file `fastsum_benchomp.c`): benchmark for multi-thread version of the NFFT-based fast summation, which outputs speedup plots (as `pgfplots`) into the LaTeX file `fastsum_benchomp_results_plots.tex`, see section 5.3 for example plots

2 Requirements

Generally, for compiling and using the multi-thread OpenMP version of the NFFT3 library, all the software libraries/tools as for building the non-parallel version of the NFFT3 library are required. In addition, the FFTW library [2] version 3 with multi-thread support and a C compiler with OpenMP 2.0 [7] support have to be available.

The multi-thread OpenMP version of the NFFT3 library was tested with FFTW library version 3.3 and GCC 4.5.1, but should also work in combination with other 3.x versions of the FFTW library and other C compilers supporting OpenMP 2.0.

When building the multi-thread OpenMP version of the MATLAB interface, a recent MATLAB version is required. Thereby, it it strongly recommended to use MATLAB R2012a or newer.

3 Usage

3.1 Compiling the NFFT3 library with OpenMP support

For detailed information about compiling (and installing) the NFFT3 library, we refer to the NFFT3 documentation [4].

The OpenMP support for the NFFT3 library can be enabled by running the `configure` script with the option `--enable-openmp`. If detailed benchmarks are going to be performed, the options `--enable-measure-time` and `--enable-measure-time-fftw` should also be added.

Then, the library can be compiled using the `make` command and installed using the `make install` command. Afterwards, two versions of the NFFT3 library are available:

- the standard (non-parallel) version (`libnfft3.so` and/or `libnfft3.a`),
- the multi-thread OpenMP version (`libnfft3_threads.so` and/or `libnfft3_threads.a`).

3.2 Using the NFFT3 library with OpenMP support in applications

Due to the identical API of the non-parallel and multi-thread version of the NFFT3 library, applications that already use the non-parallel version can easily switch to the multi-thread version:

1. The application must be linked against the multi-thread OpenMP version of the NFFT3 library (`libnfft3_threads.so` or `libnfft3_threads.a`) and the multi-thread version of the FFTW version 3. For instance, the linking arguments may be: `-lnfft3_threads -lfftw3_threads -lfftw3`
2. At the beginning of the application's main function, the FFTW function `fftw_init_threads()` should be called as described in section 5 of the FFTW documentation [2].
3. If the application contains time-intensive (non-parallel) computation steps, these computation steps should be parallelized as well.

If you intend to use the FFTW itself in your application, please note that **only FFTW plan execution is thread-safe** as described in the FFTW manual! Do **never** call any other FFTW functions, such as plan creation, from multi-threaded code parts of your application as this will likely result in segmentation faults, incorrect results and strange behaviour!

As mentioned in section 1, examples which utilize the multi-thread OpenMP version of the NFFT3 library are:

- `examples/nfft/simple_test_threads` (file `simple_test_threads.c`)
- `examples/nfft/nfft_benchomp` (file `nfft_benchomp.c`)
- `examples/nfsft/simple_test_threads` (file `simple_test_threads.c`)
- `examples/nfsft/nfsft_benchomp` (file `nfsft_benchomp.c`)
- `applications/fastsum/fastsum_test_threads` (files `fastsum_test.c`, `fastsum.c`)
- `applications/fastsum/fastsum_benchomp` (file `fastsum_benchomp.c`)

None of these examples/applications is built unless compiling the OpenMP version has been enabled (cf. section 3.1). Furthermore, both NFSFT examples are only compiled if the nfsft kernel has been enabled (`configure` switch `--enable-nfsft` or `--enable-all`).

Please note that the `configure` switch `--enable-all` does not enable the OpenMP support.

3.3 Compiling and using the MATLAB interface of the NFFT3 library

The MATLAB interface of the NFFT3 library currently supports the nfft and nfsft kernel. It consists of two MATLAB Executable files (MEX-files), one for the each kernel.

Generally, for building the MATLAB interface, the option `--with-matlab=MATLAB_DIR` has to be passed to the `configure` script, where `MATLAB_DIR` is the directory of the MATLAB installation. If the MATLAB interface for the nfsft kernel should also be built, the option `--enable-nfsft` or `--enable-all` has to be added. As mentioned before, the `configure` switch `--enable-all` does not enable the OpenMP support.

In order to build the multi-thread OpenMP version of the MATLAB interface instead of the non-parallel version, the option `--enable-openmp` has to be added, i.e. the NFFT3 library has to be built with OpenMP support.

An example for compiling and installing the MATLAB interface of the NFFT3 library for the nfft and nfsft kernel with OpenMP support, where the installation directory is `/usr/local` and the MATLAB directory is `/usr/shared/packages/matlabr2012a`, may be:

1. `./configure --prefix=/usr/local --enable-nfsft --enable-openmp`
   `--with-matlab=/usr/shared/packages/matlabr2012a`
2. `make`
3. `make install`

When running the `make install` command, the NFFT3 library files and the MATLAB interface files are installed in the directory `INSTALL_DIR/lib`, where `INSTALL_DIR` is the installation directory (`/usr/local` per default if the `configure` option `--prefix` has not been specified). MATLAB scripts demonstrating the usage of the MATLAB interface are installed in `INSTALL_DIR/share/nfft/matlab/nfft` for the nfft kernel and in `INSTALL_DIR/share/nfft/matlab/nfsft` for the nfsft kernel.

For running the example MATLAB scripts, the following steps can be completed:

1. Start MATLAB.
2. Run the command: `addpath INSTALL_DIR/lib`
   where `INSTALL_DIR` is the installation directory as describe above
3. Change either to the
   `INSTALL_DIR/share/nfft/matlab/nfft` or
   `INSTALL_DIR/share/nfft/matlab/nfsft` directory (in MATLAB)
4. Execute the MATLAB script `simple_test`

4 Implementation details

4.1 OpenMP parallelization in general

Most of the OpenMP parallelization in the NFFT3 library was performed using the OpenMP work-sharing construct `omp for` (see [7, section 2.4.1]). In order to apply the construct `omp for` without further helper constructs (e.g. synchronization mechanisms), certain conditions should be fulfilled:

- One for loop with one loop variable should exist and should increment/decrement this variable between a lower and upper boundary. Alternatively in case of several nested for loops, this should apply to the outer for loop.
- Two distinct (outer) loop iterations should not update or write to one and the same value / array entry.
- Temporary variables defined outside the (outer) for loop should only be scalars.
- The final result should not depend on the order in which the loop iterations are executed. Especially, results in one loop iteration should not depend on temporary variables / results from another loop iteration.

4.2 nfft kernel

4.2.1 Notation and NFFT algorithm

Subsequently, we give a brief summary of the description of the nonequidistant fast Fourier transform (NFFT) and its adjoint version ($\mathrm{NFFT}^{\mathsf{H}}$) from [4, appendix B]. Let $d \in \mathbb{N}$ be the dimensionalilty, $\boldsymbol{N} := (N_0, \ldots, N_{d-1})^\top \in 2\mathbb{N}^d$ the multibandlimit and $I_{\boldsymbol{N}} := \{-\frac{N_0}{2}, \ldots, \frac{N_0}{2} - 1\} \times \ldots \times \{-\frac{N_{d-1}}{2}, \ldots, \frac{N_{d-1}}{2} - 1\}$ a multi-index set with cardinality $|I_{\boldsymbol{N}}| = \prod_{t=0}^{d-1} N_t$. We consider the NFFT, i.e. the fast evaluation of the trigonometric polynomial

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi\mathrm{i}\boldsymbol{k}\boldsymbol{x}}$$

at nonequispaced (sampling) nodes $\boldsymbol{x}_j := (x_{j,0}, \ldots, x_{j,d-1})^\top \in \mathbb{T}^d$, $j = 0, \ldots, M-1$, for given Fourier coefficients $\hat{f}_{\boldsymbol{k}} \in \mathbb{C}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$,

$$f_j := f(\boldsymbol{x}_j) = \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi\mathrm{i}\boldsymbol{k}\boldsymbol{x}_j}, \qquad j = 0, \ldots, M-1, \tag{4.1}$$

as well as the $\mathrm{NFFT}^{\mathsf{H}}$, i.e. the fast evaluation of the adjoint problem

$$\hat{h}_{\boldsymbol{k}} := h(\boldsymbol{k}) = \sum_{j=0}^{M-1} f_j \mathrm{e}^{2\pi\mathrm{i}\boldsymbol{k}\boldsymbol{x}_j}, \qquad \boldsymbol{k} \in I_{\boldsymbol{N}}, \tag{4.2}$$

for given coefficients $f_j \in \mathbb{C}$, $j = 0, \ldots, M-1$.

Let $\sigma > 1$ be the oversampling factor, $\boldsymbol{n} := (n_0, \ldots, n_{d-1})^\top = \sigma\boldsymbol{N} \in 2\mathbb{N}^d$ the oversampled multibandlimit, $I_{\boldsymbol{n}} := \{-\frac{n_0}{2}, \ldots, \frac{n_0}{2} - 1\} \times \ldots \times \{-\frac{n_{d-1}}{2}, \ldots, \frac{n_{d-1}}{2} - 1\}$ a multi-index set with cardinality $|I_{\boldsymbol{n}}| = \prod_{t=0}^{d-1} n_t$, m the cut-off parameter,

$$I_{\boldsymbol{n},m}(\boldsymbol{x}_j) := \{\boldsymbol{l} \in I_{\boldsymbol{n}} : \ \boldsymbol{n} \odot \boldsymbol{x}_j - m\mathbf{1}_d \le \boldsymbol{l} \le \boldsymbol{n} \odot \boldsymbol{x}_j + m\mathbf{1}_d\}$$

```
Input: $d$, $M \in \mathbb{N}$, $\boldsymbol{N} \in 2\mathbb{N}^d$, $\boldsymbol{x}_j \in [-\frac{1}{2}, \frac{1}{2})^d$, $j = 0, \ldots, M-1$, and $\hat{f}_{\boldsymbol{k}} \in \mathbb{C}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$.

1: For $\boldsymbol{k} \in I_{\boldsymbol{N}}$ compute $\hat{g}_{\boldsymbol{k}} := |I_{\boldsymbol{n}}|^{-1} \cdot \hat{f}_{\boldsymbol{k}} / c_{\boldsymbol{k}}(\tilde{\varphi})$.
2: For $\boldsymbol{l} \in I_{\boldsymbol{n}}$ compute $g_{\boldsymbol{l}} := \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{g}_{\boldsymbol{k}} \, \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}(\boldsymbol{n}^{-1} \odot \boldsymbol{l})}$ by a $d$-variate FFT.
3: For $j = 0, \ldots, M-1$ compute $f_j := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}} \, \tilde{\varphi}(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l})$.

Output: Approximate function values $f_j$, $j = 0, \ldots, M-1$.
Arithmetic cost: $|I_{\boldsymbol{N}}| + |I_{\boldsymbol{n}}| \log |I_{\boldsymbol{n}}| + 2(2m+1)^d M$ + evaluations of the window function.
```

Algorithm 1: Nonequispaced fast Fourier transform (NFFT).

```
Input: $d$, $M \in \mathbb{N}$, $\boldsymbol{N} \in 2\mathbb{N}^d$, $\boldsymbol{x}_j \in [-\frac{1}{2}, \frac{1}{2})^d$, and $f_j \in \mathbb{C}$, $j = 0, \ldots, M-1$.

1: For $\boldsymbol{l} \in I_{\boldsymbol{n}}$ compute $g_{\boldsymbol{l}} := \sum_{j \in I^{\top}_{\boldsymbol{n},m}(\boldsymbol{l})} f_j \, \tilde{\varphi}(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l})$.
2: For $\boldsymbol{k} \in I_{\boldsymbol{N}}$ compute $\hat{g}_{\boldsymbol{k}} := \sum_{\boldsymbol{l} \in I_{\boldsymbol{n}}} g_{\boldsymbol{l}} \, \mathrm{e}^{+2\pi \mathrm{i} \boldsymbol{k}(\boldsymbol{n}^{-1} \odot \boldsymbol{l})}$ by $d$-variate (backward) FFT.
3: For $\boldsymbol{k} \in I_{\boldsymbol{N}}$ compute $\hat{h}_{\boldsymbol{k}} := |I_{\boldsymbol{n}}|^{-1} \cdot \hat{g}_{\boldsymbol{k}} / c_{\boldsymbol{k}}(\tilde{\varphi})$.

Output: Approximate coefficients $\hat{h}_{\boldsymbol{k}}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$.
Arithmetic cost: $|I_{\boldsymbol{N}}| + |I_{\boldsymbol{n}}| \log |I_{\boldsymbol{n}}| + 2(2m+1)^d M$ + evaluation of the window function.
```

Algorithm 2: Adjoint nonequispaced fast Fourier transform (adjoint NFFT, NFFT$^{\mathsf{H}}$).

an index set and

$$I^{\top}_{\boldsymbol{n},m}(\boldsymbol{l}) := \{ j = 0, \ldots, M-1 : \; \boldsymbol{l} - m\boldsymbol{1}_d \leq \boldsymbol{n} \odot \boldsymbol{x}_j \leq l + m\boldsymbol{1}_d \}$$

its transposed index set. Furthermore, let a window function $\varphi : \mathbb{R}^d \to \mathbb{R}^d$ be given, such that its one-periodic version $\tilde{\varphi}(\boldsymbol{x}) := \sum_{\boldsymbol{r} \in \mathbb{Z}^d} \varphi(\boldsymbol{x} + \boldsymbol{r})$ has a uniformly convergent Fourier series $\tilde{\varphi}(\boldsymbol{x}) = \sum_{\boldsymbol{k} \in \mathbb{Z}^d} c_{\boldsymbol{k}}(\tilde{\varphi}) \, \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}}$ with the Fourier coefficients

$$c_{\boldsymbol{k}}(\tilde{\varphi}) := \int_{\mathbb{T}^d} \tilde{\varphi}(\boldsymbol{x}) \, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}} \, \mathrm{d}\boldsymbol{x} = \int_{\mathbb{R}^d} \varphi(\boldsymbol{x}) \, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}} \, \mathrm{d}\boldsymbol{x} =: \hat{\varphi}(\boldsymbol{k}), \qquad \boldsymbol{k} \in \mathbb{Z},$$

and is well localized in the time/spatial domain $\mathbb{T}^d$ and in the frequency domain $\mathbb{Z}^d$. We define the truncated version ψ of the window function φ with $\operatorname{supp} \psi = \bigtimes_{t=0}^{d-1} [-\frac{m}{n_t}, \frac{m}{n_t}]$ by

$$\psi(\boldsymbol{x}) := \begin{cases} \varphi(\boldsymbol{x}) & \boldsymbol{x} \in \bigtimes_{t=0}^{d-1} [-\frac{m}{n_t}, \frac{m}{n_t}], \\ 0 & \text{else}, \end{cases}$$

and the corresponding one-periodic version by $\tilde{\psi}(\boldsymbol{x}) := \sum_{\boldsymbol{r} \in \mathbb{Z}^d} \psi(\boldsymbol{x} + \boldsymbol{r})$.

As described in [4, appendix B], the trigonometric sums (4.1) and (4.2) can be evaluated fast using algorithm 1 (NFFT) and 2 (NFFT$^{\mathsf{H}}$), respectively. The NFFT consists of three major steps and these steps can be written in matrix-vector-notation in the form $\boldsymbol{A}\hat{\boldsymbol{f}} = \boldsymbol{B}\boldsymbol{F}\boldsymbol{D}\hat{\boldsymbol{f}}$. Thereby, $\boldsymbol{D} \in \mathbb{R}^{|I_{\boldsymbol{n}}| \times |I_{\boldsymbol{N}}|}$ (step 1) is a "diagonal" matrix defined by

$$\boldsymbol{D} := \bigotimes_{t=0}^{d-1} \left(\boldsymbol{O}_t \mid \operatorname{diag}(1/c_{k_t}(\tilde{\varphi}))_{k_t \in I_{N_t}} \mid \boldsymbol{O}_t \right)^{\top}$$

with zero matrices $\boldsymbol{O}_t$ of size $N_t \times \frac{n_t - N_t}{2}$, $\boldsymbol{F} \in \mathbb{C}^{|I_{\boldsymbol{n}}| \times |I_{\boldsymbol{n}}|}$ (step 2) is the ordinary Fourier matrix,

$$\boldsymbol{F} := \left(\mathrm{e}^{-2\pi \boldsymbol{k}(\boldsymbol{N}^{-1} \odot \boldsymbol{j})} \right)_{\boldsymbol{j}, \boldsymbol{k} \in I_{\boldsymbol{n}}},$$

and $\boldsymbol{B} \in \mathbb{R}^{|M| \times |I_{\boldsymbol{n}}|}$ (step 3) is a sparse matrix with at most $\left(2(m+2)^d M\right)$ non-zero entries,

$$\boldsymbol{B} := \left(\tilde{\psi}(\boldsymbol{x}_j - \boldsymbol{n}^{-1} \odot \boldsymbol{l}) \right)_{j=0,\ldots,M-1; \, \boldsymbol{l} \in I_{\boldsymbol{n}}}.$$

Type	Name	Size	Description
`int`	`d`	1	Spatial dimension d
`int*`	`N`	d	Multibandwidth $\boldsymbol{N}$
`int`	`N_total`	1	Number of coefficients $\lvert I_{\boldsymbol{N}}\rvert$
`int`	`M_total`	1	Number of nodes M
`double complex*`	`f_hat`	$\lvert I_{\boldsymbol{N}}\rvert$	Fourier coefficients $\hat{\boldsymbol{f}}$ or adjoint coefficients $\hat{\boldsymbol{h}}$
`double complex*`	`f`	M	Samples $\boldsymbol{f}$
`double*`	`x`	dM	Sampling set $\mathcal{X} := \{\boldsymbol{x}_j\}_{j=0}^{M-1}$

Table 4.1: Most important members of the structure `nfft_plan` from [4, section 3].

Correspondingly, the NFFT$^{\mathsf{H}}$ consists of three major steps and these steps can be written in matrix-vector-notation in the form $\boldsymbol{A}^{\mathsf{H}}\boldsymbol{f} = \boldsymbol{D}^{\top}\boldsymbol{F}^{\mathsf{H}}\boldsymbol{B}^{\top}\boldsymbol{f}$.

Table 4.1 shows the most important members of the data structure `nfft_plan`, accessed by the user.

4.2.2 Flag `NFFT_SORT_NODES`

In the non-parallel and parallel version of the NFFT3 library, the `nfft_plan` flag `NFFT_SORT_NODES` was added. If set, the sampling nodes `x[j]`, $j = 0, \ldots, M-1$, are internally sorted as described subsequently, which can result in a performance increase as observed in the benchmarks in section 5.1.

Therefor, the pointer `int *index_x` was added to the data structure `nfft_plan` in the header file `include/nfft3.h`. When calling the initializing the NFFT/NFFT$^{\mathsf{H}}$ with the flag `NFFT_SORT_NODES` set, an `int` array of size $(2\,M)$ is allocated (but not yet initialized with values) and the pointer `index_x` is set to the beginning of this array. The array `index_x` will contain alternately a (non-unique) sort key $\in \{0, \ldots, \lvert I_{\boldsymbol{n}}\rvert - 1\}$ and a unique index $\in \{0, \ldots, M-1\}$ which refers to exactly one sampling node $\boldsymbol{x}_j$, $j = 0, \ldots, M-1$.

When the internal function `static void nfft_sort_nodes(const nfft_plan *ths)` is called, it first checks if the flag `NFFT_SORT_NODES` is set in the given data structure `nfft_plan`. If positive, the array `index_x` is initialized as follows. The sort key `index_x[2*j]`, $j = 0, \ldots, M-1$, is the lowest index

$$\boldsymbol{u}_j := (u_{j,0}, \ldots, u_{j,d-1})^{\top} := \lfloor \boldsymbol{n} \odot \boldsymbol{x}_j - m\boldsymbol{1}_d \rfloor$$

of the multi-index set $I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$ in the linearized form

$$\texttt{index\_x[2*j]} = (\ldots(((u_{j,0}\, n_1) + u_{j,1})\, n_2 + u_{j,2})\ldots)\, n_{d-1} + u_{j,d-1}$$

(row-major indexing). The unique indices `index_x[2*j+1]` refering to the sampling nodes $\boldsymbol{x}_j$, $j = 0, \ldots, M-1$, are set `index_x[2*j+1]=j`. Then, the array is sorted by the sort key using a least significant digit radix sort [5] which reorders the sort key along with the corresponding unique index refering to a sampling node.

In the third step of the NFFT (multiplication with matrix $\boldsymbol{B}$ in matrix-vector-notation), the result values `f[j]`$=f(\boldsymbol{x}_j)$ are evaluated in a different order,

```
for $k = 0 \to (M-1)$ do
    $j \leftarrow$ index_x$[2 \cdot k + 1]$
    f$[j] \leftarrow \sum_{\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}}\, \varphi(\boldsymbol{x}_j - \boldsymbol{l} \odot \boldsymbol{n}^{-1})$
end for
```

instead of

```
for $j = 0 \to (M-1)$ do
    f$[j] \leftarrow \sum_{\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}}\, \varphi(\boldsymbol{x}_j - \boldsymbol{l} \odot \boldsymbol{n}^{-1})$
end for
```

Thereby, the (inner) summation order when calculating `f[j]` remains unchanged, such that the numerical results are identical regardless of whether the flag `NFFT_SORT_NODES` is set or not.

Correspondingly, in the first step of the NFFT$^{\mathsf{H}}$ (multiplication with matrix $\boldsymbol{B}^{\top}$ in matrix-vector-notation), the new evaluation order of $g_{\boldsymbol{l}}$, $\boldsymbol{l} \in I_{\boldsymbol{n}}$, is

```
for $k = 0 \to (M-1)$ do
  $j \leftarrow$ index_x$[2 \cdot k + 1]$
  for $\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$ do
    $g_{\boldsymbol{l}} \leftarrow g_{\boldsymbol{l}}$+f$[j]$ $\varphi(\boldsymbol{x}_j - l \odot \boldsymbol{n}^{-1})$
  end for
end for
```

instead of

```
for $j = 0 \to (M-1)$ do
  for $\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$ do
    $g_{\boldsymbol{l}} \leftarrow g_{\boldsymbol{l}}$+f$[j]$ $\varphi(\boldsymbol{x}_j - l \odot \boldsymbol{n}^{-1})$
  end for
end for
```

This modification results in a different summation order, which typically leads to (slightly) different numerical results due to the usage of floating point numbers. As mentioned in section 5.1.3 and as it is noticeable in the plots in section 5.1.2 and 5.1.3, this modified summation order can result a distinct reduction of the computation time for both the non-parallel and the OpenMP version of the NFFT/NFFT$^{\mathsf{H}}$ due to CPU cache effects.

If the `nfft_plan` is initialized with one of the flags `PRE_PSI`, `PRE_FULL_PSI` or `PRE_FG_PSI`, the function `nfft_sort_nodes(nfft_plan *)` will be called from the corresponding precompute function `nfft_precompute_psi(nfft_plan *)`, `nfft_precompute_full_psi(nfft_plan *)` and `nfft_precompute_fg_psi(nfft_plan *)`, respectively. Otherwise, the function `nfft_sort_nodes(nfft_plan *)` will be called each time the NFFT (function `nfft_trafo(nfft_plan *)`) or NFFT$^{\mathsf{H}}$ (function `nfft_adjoint(nfft_plan *)`) is executed.

4.2.3 OpenMP parallelization NDFT and NDFT$^{\mathsf{H}}$

The NDFT, i.e. the direct evaluation of the trigonometric sums (4.1),

$$f_j := f(\boldsymbol{x}_j) = \sum_{\boldsymbol{k} \in I_{\boldsymbol{N}}} \hat{f}_{\boldsymbol{k}} \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k} \boldsymbol{x}_j}, \qquad j = 0, \ldots, M-1,$$

and the NDFT$^{\mathsf{H}}$, i.e. the direct evaluation of the trigonometric sums (4.2),

$$\hat{h}_{\boldsymbol{k}} := h(\boldsymbol{k}) = \sum_{j=0}^{M-1} f_j \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k} \boldsymbol{x}_j}, \qquad \boldsymbol{k} \in I_{\boldsymbol{N}},$$

are available through the functions `nfft_trafo_direct(nfft_plan *ths)` and `nfft_adjoint_direct(nfft_plan *ths)`, respectively. Thereby, both functions are implemented using macros, the NDFT via the macro `MACRO_ndft(trafo)` and the NDFT$^{\mathsf{H}}$ via the macro `MACRO_ndft(adjoint)`. For the OpenMP parallelization, these two macro calls were unrolled.

NDFT

The NDFT function `nfft_trafo_direct(nfft_plan *ths)` contains a distinction of cases for the one- and multi-dimensional case. In both cases, there exists an outer loop

```
for (j = 0; j < ths->M_total; j++)
```

which considers each sampling node $\boldsymbol{x}_j$, calculates the value $f(\boldsymbol{x}_j)$ and writes the resulting value to `ths->f[j]`. The statements inside the for loop do not write to variables outside the outer for loop except `ths->f[j]`. Therefore, the for loop can be parallelized by adding the OpenMP pragma

```
#pragma omp parallel for default(shared) private(j)
```

and moving all temporary variable definitions inside the outer for loop.

NDFT$^{\mathsf{H}}$

Similarly to the NDFT implementation, the NDFT$^{\mathsf{H}}$ function `nfft_trafo_direct(nfft_plan *ths)` contains the distinction of cases and an outer for loop over the sampling nodes $\boldsymbol{x}_j$. In contrast to before, several loop iterations may update one and the same result entry `ths->f_hat[k]`, which corresponds to $\hat{h}_{\boldsymbol{k}}$. For example, the one-dimensional case reads as follows:

```
int j;
for (j = 0; j < ths->M_total; j++)
{
  int k_L;
  for(k_L = 0; k_L < ths->N_total; k_L++)
  {
    double omega = (k_L - (ths->N_total/2)) * K2PI * ths->x[j];
    ths->f_hat[k_L] += ths->f[j]*cexp(+( 1.0iF)*omega);
  }
}
```

Furthermore, in the multi-dimensional case, intermediate results are used, i.e. the statements inside the outer for loop use temporary variables which change from one outer loop iteration to the next one and depend on the previous iteration. Each of these two issues causes a problem for the OpenMP parallelization.

In both the one- and multi-dimensional case, the inner and outer loops are exchanged for the OpenMP parallelization, whereas the non-OpenMP code remains unchanged if the non-OpenMP version of the NDFT$^{\mathsf{H}}$ is used.

For the one-dimensional case with OpenMP enabled, the resulting code is:

```
int k_L;
#pragma omp parallel for default(shared) private(k_L)
for(k_L = 0; k_L < ths->N_total; k_L++)
{
  int j;
  for (j = 0; j < ths->M_total; j++)
  {
    double omega = (k_L - (ths->N_total/2)) * K2PI * ths->x[j];
    f_hat[k_L] += f[j]*cexp(+( 1.0iF)*omega);
  }
}
```

Furthermore, the dependencies between loop iterations have been removed in the multi-dimensional case.

4.2.4 OpenMP parallelization NFFT

As mentioned in section 4.2.1, the NFFT, which calculates a fast approximation to the problem (4.1), consists of three major steps, which can be written in matrix-vector-notation as $\boldsymbol{A}\hat{\boldsymbol{f}} = \boldsymbol{BFD}\hat{\boldsymbol{f}}$. These steps were parallelized as follows.

First step (diagonal matrix D)

The code of the first NFFT step ($\hat{g}_{\boldsymbol{k}} := \hat{f}_{\boldsymbol{k}}/c_{\boldsymbol{k}}(\tilde{\varphi})$, $k \in \boldsymbol{I_N}$) resides inside the function `nfft_trafo_1d`, `nfft_trafo_2d` and `nfft_trafo_3d` for the one-/two-/three-dimensional case and in its own function `nfft_D_A` for higher dimensions. In the one-dimensional case, there exists one for loop which can be easily parallelized.

In the two- and three-dimensional case, the code contains two and three nested loops, respectively. For instance, the two-dimensional case (without precomputing the values $\hat{\varphi}(\boldsymbol{k} \odot \boldsymbol{n}^{-1})$, $\boldsymbol{k} \in \boldsymbol{I_N}$) reads as follows:

```
for(k0 = 0; k0 < N0/2; k0++)
{
  ck01 = K(1.0)/(PHI_HUT(k0-N0/2,0));
  ck02 = K(1.0)/(PHI_HUT(k0,0));
```

```
    for(k1=0;k1<N1/2;k1++)
    {
      ck11 = K(1.0)/(PHI_HUT(k1-N1/2,1));
      ck12 = K(1.0)/(PHI_HUT(k1,1));
      g_hat[(n0-N0/2+k0)*n1+n1-N1/2+k1] = f_hat[k0*N1+k1] * ck01 * ck11;
      g_hat[k0*n1+n1-N1/2+k1] = f_hat[(N0/2+k0)*N1+k1] * ck02 * ck11;
      g_hat[(n0-N0/2+k0)*n1+k1] = f_hat[k0*N1+N1/2+k1] * ck01 * ck12;
      g_hat[k0*n1+k1] = f_hat[(N0/2+k0)*N1+N1/2+k1] * ck02 * ck12;
    }
  }
```

Due to this code structure, only the outer loop (`for (k0 ...)`) was parallelized using an OpenMP for pragma. Therefore, only up to $N_0/2$ threads can be used by OpenMP limiting the possible speedup for the current step to $N_0/2$.

In the higher dimensional case ($d > 3$), the non-OpenMP implementation uses a single loop `for (k_L = 0; k_L < ths->N_total; k_L++)`, `ths->N_total` $= |I_{\boldsymbol{N}}|$, and internally calculates and increments the multi-index $\boldsymbol{k}$, i.e. one loop iteration uses (temporary) values from the previous iteration.

Therefore, the OpenMP code was rewritten based on the existing code. Instead of incrementing the multi-index $\boldsymbol{k}$, $\boldsymbol{k}$ is re-calculated each loop iteration from the loop variable `k_L`. The advantage of this approach is clearly that the (theoretically) usable number of threads is `ths->N_total`. However, the disadvantage of this implementation is that the multi-index $\boldsymbol{k}$ has to be calculated for each loop iteration and this calculation particularly uses integer division and modulo operations, which may be more time-consuming than increment and compare operations.

The original function `nfft_D_A` containing the non-OpenMP code was renamed to `nfft_D_serial_A`, the OpenMP implementation was written to the function `nfft_D_openmp_A`. A new function `nfft_D_A` was implemented that calls either `nfft_D_serial_A` or `nfft_D_openmp_A`.

Second step (Fourier matrix F)

Since the FFTW library is used for executing this step, the multi-threaded version of the FTTW is simply used if the NFFT is compiled with OpenMP support. Thereby, only FFTW plan execution is thread-safe as described in [2, section 5].

By default, the multi-threaded FFTW will only use one thread for the FFT unless a higher number of threads is specified. For changing the number of threads to be used, the function `fftw_plan_with_nthreads(int)` has to be called before a planning function. All FFTW plans created afterwards will use the specified number of threads, whereas already created FFTW plans will use the number of threads valid at their respective creation time.

The FFT in this step uses the same number of threads as available in the first and third step. This is achieved by calling the following helper function

```
int nfft_get_omp_num_threads()
{
  int nthreads_return_value;
  #pragma omp parallel default(shared)
  {
    int nthreads = omp_get_num_threads();
    #pragma omp master
    {
      nthreads_return_value = nthreads;
    }
  }
  return nthreads_return_value;
}
```

FFTW plan creation in the internal NFFT function `nfft_init_help` uses a critical section to set the number of FFTW threads and create the internal FFTW plans. This ensures that the init functions of the nfft kernel can be called concurrently from several threads. In particular, it is possible to concurrently initialize and run several nfft plans with a different number of threads.

The code in the internal NFFT function `nfft_init_help` looks as follows.

```
#ifdef _OPENMP
{
  int nthreads = nfft_get_omp_num_threads();

  #pragma omp critical (nfft_omp_critical_fftw_plan)
  {
    fftw_plan_with_nthreads(nthreads);
    ths->my_fftw_plan1 = fftw_plan_dft(ths->d, ths->n, ths->g1, ths->g2,
                                       FFTW_FORWARD, ths->fftw_flags);
    ths->my_fftw_plan2 = fftw_plan_dft(ths->d, ths->n, ths->g2, ths->g1,
                                       FFTW_BACKWARD, ths->fftw_flags);
  }
}
#else
    ths->my_fftw_plan1 = fftw_plan_dft(ths->d, ths->n, ths->g1, ths->g2,
                                       FFTW_FORWARD, ths->fftw_flags);
    ths->my_fftw_plan2 = fftw_plan_dft(ths->d, ths->n, ths->g2, ths->g1,
                                       FFTW_BACKWARD, ths->fftw_flags);
#endif
```

Third step (sparse matrix B)

The code for the third step of the NFFT ($f(\boldsymbol{x}_j) = \sum_{\boldsymbol{l}\in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)} g_{\boldsymbol{l}}\, \varphi(\boldsymbol{x}_j - \boldsymbol{l} \odot \boldsymbol{n}^{-1})$, $j = 0, \ldots, M-1$) resides in the functions `nfft_trafo_1d_B`, `nfft_trafo_2d_B`, `nfft_trafo_3d_B` and `nfft_B_A` for the one-/two-/three-dimensional case and for the case $d \geq 4$, respectively. In each of these functions, the implementation has a distinction of cases for the NFFT flags `PRE_FULL_PSI`, `PRE_PSI`, `PRE_FG_PSI`, `FG_PSI`, `PRE_LIN_PSI` and if none of the aforementioned flags is set. In each case, there exists an outer loop of the form

```
for (k = 0; k < M; k++)
{
  int j = (ths->nfft_flags & NFFT_SORT_NODES) ? ths->index_x[2*k+1] : k;

  ...
}
```

which iterates over the sampling nodes $\boldsymbol{x}_j$ and which contains one or several inner loop(s) iterating over the index set $I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$. Thereby, the computation in one outer loop iteration does not depend on another one. Therefore, the definitions of all (temporary) variables used inside the outer loop were moved from the beginning of the functions into the outer loop and then, the outer loop could be simply parallelized by adding the OpenMP pragma `#pragma omp parallel for default(shared) private(k)` in front.

4.2.5 OpenMP parallelization NFFT$^{\mathsf{H}}$

The NFFT$^{\mathsf{H}}$ consists of three major steps, $\boldsymbol{A}^{\mathsf{H}}\boldsymbol{f} = \boldsymbol{D}^{\top}\boldsymbol{F}^{\mathsf{H}}\boldsymbol{B}^{\top}\boldsymbol{f}$, as described in section 4.2.1. These steps were parallelized similar to the NFFT with special attention to the first calculation step.

First step (sparse matrix $\mathbf{B}^{\top}$)

The code for this step resides in the functions `nfft_adjoint_1d_B`, `nfft_adjoint_2d_B`, `nfft_adjoint_3d_B` and `nfft_B_T` for the cases $d = 1$, $d = 2$, $d = 3$ and $d \geq 4$, respectively.

Similarly, to the original implementation of the NDFT$^{\mathsf{H}}$ as described in section 4.2.3, there exists an outer loop iterating over the the sampling nodes $\boldsymbol{x}_j$ and several loop iterations may update one and the same entry in the array ths->g of the corresponding nfft_plan *ths. In pseudo code, this step calculates:

```
for $j = 0 \to (M-1)$ do
    for $\boldsymbol{l} \in I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$ do
        $g_{\boldsymbol{l}} \leftarrow g_{\boldsymbol{l}}$+f[$j$] $\varphi(\boldsymbol{x}_j - l \odot \boldsymbol{n}^{-1})$
    end for
end for
```

However, it is problematic to exchange the inner and outer loops this time, since determining the adjoint $I^{\top}_{\boldsymbol{n},m}(\boldsymbol{l}) := \{j = 0, \dots, M-1 :\ \boldsymbol{l} - m\mathbf{1}_d \le \boldsymbol{n} \odot \boldsymbol{x}_j \le l + m\mathbf{1}_d\}$ of the index set $I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$ is more difficult.

One simple modification is to parallelize the outer loop iterating over the sampling nodes $\boldsymbol{x}_j$, while ensuring that the update operations to the array ths->g from within the inner loops are performed atomically. This can be achieved using the OpenMP statement atomic which creates a critical section for the updates.

Another implementation is available via the nfft_plan flag NFFT_OMP_BLOCKWISE_ADJOINT. If set, the array ths->g (values $g_{\boldsymbol{l}}$) is "distributed" among the threads, i.e. each thread is responsible for updating a (continuous) portion of the array ths->g.

Therefor, $g_{\boldsymbol{l}}$ is (almost) uniformly divided along its first dimension. Let $T \in \mathbb{N}$ be the number of OpenMP threads, $i^{(r)}_{\mathrm{u}_0} \in \{0, \dots, n_0 - 1\}$ the lower index boundary (in the first dimension) of thread r and $i^{(r)}_{\mathrm{o}_0} \in \{0, \dots, n_0 - 1\}$ the upper index boundary (in the first dimension) of thread r, whereat

$$i^{(r)}_{\mathrm{o}_0} - i^{(r)}_{\mathrm{u}_0} \in \{\lfloor n_0/T \rfloor, \lfloor n_0/T \rfloor + 1\} \quad (r \in \{0, \dots, T-1\}),$$

$$i^{(r)}_{\mathrm{u}_0} > i^{(r-1)}_{\mathrm{o}_0} \quad (r \in \{1, \dots, T-1\}),$$

$$\sum_{r=0}^{T-1} i^{(r)}_{\mathrm{o}_0} - i^{(r)}_{\mathrm{u}_0} = n_0, \quad i^{(0)}_{\mathrm{u}_0} := 0, \quad i^{(T-1)}_{\mathrm{o}_0} := n_0 - 1.$$

Then, thread r, $r \in \{0, \dots, T-1\}$, (and only this thread) is responsible for calculating the coefficients

$$g_{\boldsymbol{l}'}, \quad \boldsymbol{l}' := (l'_0, \dots, l'_{d-1})^{\top} \in I_{\boldsymbol{n}} \ \cap \ \left(\{i^{(r)}_{\mathrm{u}_0}, \dots, i^{(r)}_{\mathrm{o}_0}\} \times \mathbb{Z}^{d-1}\right),$$

and for updating the entries ths->g[$i^{(r)}_{\mathrm{u}_0} \prod_{t=1}^{d-1} n_t \ \dots \ (i^{(r)}_{\mathrm{o}_0} + 1)(\prod_{t=1}^{d-1} n_t) - 1$]. Each thread has to determine the corresponding sampling nodes

$$\boldsymbol{x}_{j'},\ j' \in I_{\mathcal{X}^{(r)}} := \bigcup_{\substack{\boldsymbol{l}' := (l'_0, \dots, l'_{d-1})^{\top} \in I_{\boldsymbol{n}} \\ l'_0 \in \{(i^{(r)}_{\mathrm{u}_0}, \dots, i^{(r)}_{\mathrm{o}_0}\}}} I^{\top}_{\boldsymbol{n},m}(\boldsymbol{l}').$$

Using the definition of the lowest index

$$\boldsymbol{u}_j := (u_{j,0}, \dots, u_{j,d-1})^{\top} := \lfloor \boldsymbol{n} \odot \boldsymbol{x}_j - m\mathbf{1}_d \rfloor$$

of the multi-index set $I_{\boldsymbol{n},m}(\boldsymbol{x}_j)$ from section 4.2.2, we get

$$I_{\mathcal{X}^{(r)}} = \left\{ j \in \{0, \dots, M-1\} :\ 0 \le \left((u_{j,0} - i^{(r)}_{\mathrm{u}_0}) \mod n_0 \right) \le 2m + 1 \right\}.$$

Next, we can use the sort index index_x[2*k] from section 4.2.2 and perform a binary search for the lowest index $\boldsymbol{u}_j$ in the entries of the array index_x. Let

$$\begin{aligned}
\texttt{min\_u\_a}^{(\texttt{r})} &:= \max\{0, (i^{(r)}_{\mathrm{u}_0} - 2m - 1) \prod_{t=1}^{d-1} n_t\}, \\
\texttt{max\_u\_a}^{(\texttt{r})} &:= \min\{(\prod_{t=0}^{d-1} n_t) - 1\,,\ (i^{(r)}_{\mathrm{o}_0} + 1)(\prod_{t=1}^{d-1} n_t) - 1\}, \\
\texttt{min\_u\_b}^{(\texttt{r})} &:= (i^{(r)}_{\mathrm{u}_0} - 2m - 1) \prod_{t=1}^{d-1} n_t, \\
\texttt{max\_u\_b}^{(\texttt{r})} &:= (\prod_{t=0}^{d-1} n_t) - 1,
\end{aligned}$$

$$I_{\mathcal{X}_{\mathrm{a}}^{(r)}} := \left\{k \in \{0,\dots,M-1\} : \texttt{min\_u\_a}^{(\texttt{r})} \le \texttt{index\_x}[2*\texttt{k}] \le \texttt{max\_u\_a}^{(\texttt{r})}\right\},$$

$$I_{\mathcal{X}_{\mathrm{b}}^{(r)}} := \begin{cases} \left\{k \in \{0,\dots,M-1\} : \texttt{min\_u\_b}^{(\texttt{r})} \le \texttt{index\_x}[2*\texttt{k}] \le \texttt{max\_u\_b}^{(\texttt{r})}\right\} & \text{if } 2m+1 \le i_{\mathrm{u}_0}^{(r)} \\ \emptyset & \text{otherwise.} \end{cases}$$

Then, $I_{\mathcal{X}_{\mathrm{a}}^{(r)}} \cap I_{\mathcal{X}_{\mathrm{b}}^{(r)}} = \emptyset$ and $I_{\mathcal{X}^{(r)}} = I_{\mathcal{X}_{\mathrm{a}}^{(r)}} \cup I_{\mathcal{X}_{\mathrm{b}}^{(r)}}$. We determine

$$k_{\mathrm{a}_{\min}}^{(r)} := \min\left\{\texttt{k} \in \{0,\dots,M-1\} : \texttt{min\_u\_a}^{(\texttt{r})} \le \texttt{index\_x}[2*\texttt{k}]\right\}$$

and get

$$I_{\mathcal{X}_{\mathrm{a}}^{(r)}} = \left\{\texttt{index\_x}[2*\texttt{k}+1] : k_{\mathrm{a}_{\min}}^{(r)} \le \texttt{k} \le k_{\mathrm{a}_{\max}}^{(r)}\right\}$$

with $k_{\mathrm{a}_{\max}}^{(r)} := \max\left\{\texttt{k} \in \{0,\dots,M-1\} : \texttt{index\_x}[2*\texttt{k}] \le \texttt{max\_u\_a}^{(\texttt{r})}\right\}$. In the case $I_{\mathcal{X}_{\mathrm{b}}^{(r)}} \neq \emptyset$, we also determine

$$k_{\mathrm{b}_{\min}}^{(r)} := \min\left\{k \in \{0,\dots,M-1\} : \texttt{min\_u\_b}^{(\texttt{r})} \le \texttt{index\_x}[2*\texttt{k}]\right\}.$$

and get

$$I_{\mathcal{X}_{\mathrm{b}}^{(r)}} = \left\{\texttt{index\_x}[2*\texttt{k}+1] : k_{\mathrm{b}_{\min}}^{(r)} \le \texttt{k} \le k_{\mathrm{b}_{\max}}^{(r)}\right\}$$

with $k_{\mathrm{b}_{\max}}^{(r)} := \max\left\{\texttt{k} \in \{0,\dots,M-1\} : \texttt{index\_x}[2*\texttt{k}] \le \texttt{max\_u\_b}^{(\texttt{r})}\right\}$.

Based on these definitions, the non-parallel implementation of this first step was modified for the one-/two-/three-dimensional case as follows.

If the OpenMP version is built and the `nfft_plan` flag `NFFT_OMP_BLOCKWISE_ADJOINT` is not set, an outer for loop iterating over the sampling nodes $\boldsymbol{x}_j$ is used and the sub-function `nfft_adjoint_1d_compute_omp_atomic`, `nfft_adjoint_2d_compute_omp_atomic` or `nfft_adjoint_3d_compute_omp_atomic` is called inside the outer loop (instead of `nfft_adjoint_1d_compute_serial`, `nfft_adjoint_2d_compute_serial` or `nfft_adjoint_3d_compute_serial` in the non-OpenMP version). Inside this sub-function, the entries of the array `ths->g` are updated using the OpenMP statement `atomic`.

Otherwise, if the `nfft_plan` flag `NFFT_OMP_BLOCKWISE_ADJOINT` is set, the values $i_{\mathrm{u}_0}^{(r)}$, $i_{\mathrm{o}_0}^{(r)}$, $\texttt{min\_u\_a}^{(\texttt{r})}$, $\texttt{max\_u\_a}^{(\texttt{r})}$, $\texttt{min\_u\_b}^{(\texttt{r})}$ and $\texttt{max\_u\_b}^{(\texttt{r})}$ are determined for each thread r calling the helper function `nfft_adjoint_B_omp_blockwise_init` from within an OpenMP `parallel` section and the index array `index_x` of the sampling nodes $\boldsymbol{x}_j$ is sorted as described above. Then, each (participating) thread performs a binary search for the lowest $k_{\mathrm{a}_{\min}}^{(r)}$ and, if $I_{\mathcal{X}_{\mathrm{b}}^{(r)}} \neq \emptyset$, for $k_{\mathrm{b}_{\min}}^{(r)}$. For each $\texttt{k} \in I_{\mathcal{X}_{\mathrm{a}}^{(r)}}$ and $\texttt{k} \in I_{\mathcal{X}_{\mathrm{a}}^{(r)}}$, the sub-function `nfft_adjoint_1d_compute_omp_blockwise`, `nfft_adjoint_2d_compute_omp_blockwise` or `nfft_adjoint_3d_compute_omp_blockwise` is called with the corresponding sampling node $\boldsymbol{x}_j$, $j = \texttt{index\_x}[2*\texttt{k}]$, as function argument.

For the case $d \ge 4$, the original non-OpenMP function `nfft_B_T` was renamed to `nfft_B_serial_T` and the OpenMP version was implemented inside the function `nfft_B_openmp_T`. The function `nfft_B_T` calls either `nfft_B_serial_T` for the non-OpenMP version or `nfft_B_openmp_T` for the OpenMP version of the NFFT3 library.

The implementation of `nfft_B_openmp_T` currently ignores the `nfft_plan` flag `NFFT_OMP_BLOCKWISE_ADJOINT` and only supports `atomic` updates of the entries in the array `ths->g`.

Second step (Fourier matrix $\mathbf{F}^{\mathsf{H}}$)

Since the FFTW library is used for executing this step, the FFTW is called as described for the second step of the NFFT in section 4.2.4.

Third step (diagonal matrix $\mathbf{D}^{\top}$)

For the one-/two-/three-dimensional case, the code of the last NFFT$^{\mathsf{H}}$ step resides inside the functions `nfft_adjoint_1d`, `nfft_adjoint_2d` and `nfft_adjoint_3d`, for higher dimensions in the function `nfft_D_T`.

flag	enabled	description
`PRE_PHI_HUT`	yes	pre-compute $c_{\boldsymbol{k}}(\tilde{\varphi})$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$, during initialization
`MALLOC_X`	yes	allocate array `double *x` of sampling nodes $\boldsymbol{x}_j$ during initialization
`MALLOC_F_HAT`	yes	allocate array `double complex *f_hat` of Fourier coefficients $\hat{f}_{\boldsymbol{k}}$, $\boldsymbol{k} \in I_{\boldsymbol{N}}$, during initialization
`MALLOC_F`	yes	allocate array `double complex *f` of sampling values $f(\boldsymbol{x}_j)$, $j = 0, \dots, M-1$, during initialization
`FFTW_INIT`	yes	create FFTW plans for NFFT and NFFT$^{\mathsf{H}}$ during initialization
`FFT_OUT_OF_PLACE`	yes	out-of-place FFT (different arrays for the input and output values of the FFT)
`PRE_PSI`	yes	use pre-computed values of $\psi(\boldsymbol{x_j} - \boldsymbol{n}^{-1} \odot \boldsymbol{l})$ in third step of NFFT (matrix $\boldsymbol{B}$) / first step of NFFT$^{\mathsf{H}}$ (matrix $\boldsymbol{B}^{\top}$)
`NFFT_SORT_NODES`	$d \geq 2$	sort sampling nodes as described in section 4.2.2
`NFFT_OMP_BLOCKWISE_ADJOINT`	$d \geq 2$ OpenMP	special multi-thread implementation of the first step of NFFT$^{\mathsf{H}}$ as described in section 4.2.5

Table 4.2: NFFT/NFFT$^{\mathsf{H}}$ flags set when one of the initialization functions `nfft_init_1d`, `nfft_init_2d`, `nfft_init_3d` or `nfft_init` is called.

The OpenMP parallelization was performed corresponding to the the first step of the NFFT as described in section 4.2.4. Therefore, the possible speedup of this step is limited to $N_0/2$ in the one-/two-/three-dimensional case.

The original function `nfft_D_T` containing the non-OpenMP code was renamed to `nfft_D_serial_T`, the OpenMP implementation was written to the function `nfft_D_openmp_T`. A new function `nfft_D_T` was implemented that calls either `nfft_D_serial_T` or `nfft_D_openmp_T`.

4.2.6 Default flags for NFFT and NFFT$^{\mathsf{H}}$

When the data structure `nfft_plan` is initialized using the simple API, i.e. when one of the initialization functions `nfft_init_1d`, `nfft_init_2d`, `nfft_init_3d` or `nfft_init` is called, the NFFT/NFFT$^{\mathsf{H}}$ flags `PRE_PHI_HUT`, `MALLOC_X`, `MALLOC_F_HAT`, `MALLOC_F`, `FFTW_INIT`, `FFT_OUT_OF_PLACE` and `PRE_PSI` are always set. Furthermore, in the multi-dimensional case ($d \geq 2$), the flag `NFFT_SORT_NODES` is also set. If the multi-thread OpenMP version of the NFFT3 library is built, the flag `NFFT_OMP_BLOCKWISE_ADJOINT` is additionally set in the multi-dimensional case. Table 4.2.6 summarizes the enabled flags and gives a short description of these flags.

4.3 nfsft kernel

In this section, the OpenMP parallelization of the NFSFT (nonequidistant fast spherical Fourier transform, NFFT on the sphere, cf. [9, 6]) and of the adjoint problem NFSFT$^{\mathsf{H}}$ are described. Thereby, the NFSFT denotes the fast evaluation of a function $f \in \mathrm{L}^2(\mathbb{S}^2)$ with the finite orthogonal expansion

$$f(\vartheta,\varphi) = \sum_{k=0}^{N} \sum_{n=-k}^{k} a_k^n Y_k^n(\vartheta,\varphi), \qquad N \in \mathbb{N}_0,$$

in terms of spherical harmonics Y_k^n for given spherical Fourier coefficients $a_k^n \in \mathbb{C}$ at arbitrary nodes $(\vartheta_j,\varphi_j) \in \mathbb{S}^2 := \{\boldsymbol{x} \in \mathbb{R}^3 : \|\boldsymbol{x}\|_2 = 1\}$, $j = 0,\ldots,M-1$, $M \in \mathbb{N}$, in spherical coordinates,

$$f(\vartheta_j,\varphi_j) = \sum_{k=0}^{N} \sum_{n=-k}^{k} a_k^n Y_k^n(\vartheta_j,\varphi_j), \qquad j = 0,\ldots,M-1, \tag{4.3}$$

and the NFSFT$^{\mathsf{H}}$ the fast evaluation of the sums

$$\tilde{a}_k^n := \sum_{j=0}^{M-1} f(\vartheta_j,\varphi_j)\,\overline{Y_k^n(\vartheta_j,\varphi_j)}, \qquad k = 0,\ldots,N, \quad n = -k,\ldots,k. \tag{4.4}$$

for given function values $f(\vartheta_j,\varphi_j) \in \mathbb{C}$, $j = 0,\ldots,M-1$, $M \in \mathbb{N}$.

4.3.1 OpenMP parallelization NDSFT and NDSFT$^{\mathsf{H}}$

The NDSFT, i.e. the direct evaluation of the problem (4.3), resides in the function `nfsft_trafo_direct` and the NDSFT$^{\mathsf{H}}$, i.e. the direct evaluation of the problem (4.4), resides in the function `nfsft_adjoint_direct`. Both were parallelized by adding the OpenMP pragma `#pragma omp parallel for` in front of the outer for loop, which iterates over the sampling nodes in case of the NDSFT and over the coefficients $\tilde{a}_k^n$ in case of the NDSFT$^{\mathsf{H}}$, since the calculations inside the outer for loop are independent and update distinct result entries at each loop iteration.

4.3.2 OpenMP parallelization NFSFT

The NFSFT consists of three major steps

1. $(2N+1)$ discrete/fast polynomial transforms (DPT/FPT),
2. conversion of Chebyshev coefficients to Fourier coefficients (`c2e`),
3. two-dimensional NDFT/NFFT with multibandlimit $(2N+2,\ 2N+2)^\top$,

and resides in the function `nfsft_trafo`.

DPT/FPT

Due to the $(2N+1)$ independent DPTs/FPTs called in this step, one obvious parallelization approach would be to concurrently execute the DPTs/FPTs by the available threads with minor modifications to the DPT/FPT code itself.

With the current implementation of the DPT/FPT, it is not possible to concurrently run several DPTs/FPTs with one and the same instance of the data structure `fpt_set`. Therefore, the element `fpt_set set` inside the internal NFSFT data structure `nfsft_wisdom` has to be duplicated by the number of threads and each thread has to initialize its `fpt_set` inside the function `nfsft_precompute`. As a consequence, each thread performing a DPT/FPT plans its own discrete cosine transform (DCT). The modification to the NFSFT struct `nfsft_wisdom` looks like:

```
struct nfsft_wisdom
{
    ...
#ifdef _OPENMP
```

```
    int nthreads;
    fpt_set *set_threads;
#else
    fpt_set set;
#endif
};
```

Since the DPT/FPT internally uses the FFTW for performing DCTs, it must be ensured, as mentioned in section 3.2, that FFTW plan creation/destroying functions are not concurrently executed. This is realized by enclosing all non plan-executing FFTW function in an OpenMP critical section with the identifier `nfft_omp_critical_fftw_plan`, which is the same identifier like in section 4.2.4. Although each thread plans its own DCT one after the other, the FFTW library internally performs the time-consuming parts of planning operations only once, since all DCTs are planned using identical parameters and the FFTW library internally memorizes planning results (cf. [2, section 4.3]).

c2e

This step was not parallelized, since it only requires a small part of the total computation time as benchmarks show.

2D NDFT/NFFT

The OpenMP implementation of the NDFT/NFFT from section 4.2 is called.

4.3.3 OpenMP parallelization NFSFT$^{\mathsf{H}}$

The NFSFT$^{\mathsf{H}}$ consists of three major steps

1. two-dimensional NDFT$^{\mathsf{H}}$/NFFT$^{\mathsf{H}}$ with multibandlimit $(2N+2,\ 2N+2)^\top$,
2. conversion of Fourier coefficients to Chebyshev coefficients (`c2e`$^\top$),
3. $(2N+1)$ transposed discrete/fast polynomial transforms (DPT$^\top$/FPT$^\top$),

and resides in the function `nfsft_adjoint`.

2D NDFT$^{\mathsf{H}}$/NFFT$^{\mathsf{H}}$

The OpenMP implementation of the NDFT$^{\mathsf{H}}$/NFFT$^{\mathsf{H}}$ from section 4.2 is called.

c2e$^\top$

This step was not parallelized, since it only requires a small part of the total computation time as benchmarks show.

DPT$^\top$/FPT$^\top$

The same (modified) `nfsft_wisdom` as in the first step of section 4.3.2 is used and the $(2N+1)$ adjoint DPTs/FPTs are concurrently executed by the threads.

4.4 NFFT-based fast summation (fastsum)

In this section, we consider the OpenMP parallelization of the NFFT-based fast summation (cf. [8, 1]) in `applications/fastsum`, i.e. the fast approximate evaluation of the sums

$$f(\boldsymbol{y}_j) := \sum_{\ell=0}^{L-1} \alpha_\ell \, \mathcal{K}(\boldsymbol{y}_j - \boldsymbol{x}_\ell), \quad j = 0, \dots, M-1,$$

where $\mathcal{K}\colon \mathbb{R}^d \to \mathbb{C}$ is a special kernel function, $\boldsymbol{y}_j \in \mathbb{R}^d$ are the target nodes, $\boldsymbol{x}_\ell \in \mathbb{R}^d$ the source nodes and $\alpha_\ell \in \mathbb{C}$ coefficients associated with the source nodes.

Let $\boldsymbol{N} := (N_0, \dots, N_{d-1})^\top \in 2\mathbb{N}^d$ be the multibandlimit, $\sigma > 1$ the oversampling factor, $\boldsymbol{n} := \sigma \boldsymbol{N} \in 2\mathbb{N}^d$ the oversampled multibandlimit, $p \in \mathbb{N}$ the smoothness parameter, $\varepsilon_{\mathrm{I}} < 1/4$ the near field size (/ size of the inner regularization region) and $\varepsilon_{\mathrm{B}} < 1/4$ the size of the outer regularization region. The kernel $\mathcal{K}$ is regularized near the boundaries of the d-dimensional torus $\mathbb{T}^d := [-1/2,\ 1/2)^d$ and (in case of a singularity at the origin) near the origin in order to obtain a smooth 1-periodic version $\mathcal{K}_{\mathrm{R}} \in C^{p-1}(\mathbb{R}^d)$, which is identical to the kernel $\mathcal{K}$ at $\left\{\boldsymbol{x} \in \mathbb{T}^d :\ \varepsilon_{\mathrm{I}} \le \|\boldsymbol{x}\|_2 \le 1/2 - \varepsilon_{\mathrm{B}}\right\}$. Therefor, a two-point Taylor interpolation (cf. [1, section V.2.2]) of degree p, i.e. a polynomial of at most degree $(2p-1)$, is used for the inner and outer regularization region, respectively. Then, the regularized kernel $\mathcal{K}_{\mathrm{R}}$ is approximated by the trigonometric polynomial

$$\mathcal{K}_{\mathrm{RF}}(\boldsymbol{x}) := \sum_{\boldsymbol{k} \in I_{\boldsymbol{n}}} b_{\boldsymbol{k}} \, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}}$$

with the Fourier coefficients $b_{\boldsymbol{k}} \in \mathbb{C}$ given by

$$b_{\boldsymbol{k}} := \frac{1}{|I_{\boldsymbol{n}}|} \sum_{\boldsymbol{l} \in I_{\boldsymbol{n}}} \mathcal{K}_{\mathrm{R}}\left(\boldsymbol{l} \odot \boldsymbol{n}^{-1}\right) \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\left(\boldsymbol{l} \odot \boldsymbol{n}^{-1}\right)}, \quad \boldsymbol{k} \in I_{\boldsymbol{n}}. \tag{4.5}$$

Since $\mathcal{K} = \mathcal{K}_{\mathrm{RF}} + (\mathcal{K} - \mathcal{K}_{\mathrm{R}}) + (\mathcal{K}_{\mathrm{R}} - \mathcal{K}_{\mathrm{RF}}) \approx \mathcal{K}_{\mathrm{RF}} + (\mathcal{K} - \mathcal{K}_{\mathrm{R}})$, we obtain

$$\begin{aligned}
f(\boldsymbol{y}_j) &\approx \sum_{\ell=0}^{L-1} \alpha_\ell \, \mathcal{K}_{\mathrm{RF}}(\boldsymbol{y}_j - \boldsymbol{x}_\ell) && + \sum_{\ell=0}^{L-1} \alpha_\ell \, (\mathcal{K} - \mathcal{K}_{\mathrm{R}})(\boldsymbol{y}_j - \boldsymbol{x}_\ell) \\
&= \sum_{\ell=0}^{L-1} \alpha_\ell \sum_{\boldsymbol{k} \in I_{\boldsymbol{n}}} b_{\boldsymbol{k}} \, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}(\boldsymbol{y}_j - \boldsymbol{x}_\ell)} && + \sum_{\substack{\ell=0 \\ \|\boldsymbol{y}_j - \boldsymbol{x}_\ell\|_2 < \varepsilon_{\mathrm{I}}}}^{L-1} \alpha_\ell \, (\mathcal{K} - \mathcal{K}_{\mathrm{R}})(\boldsymbol{y}_j - \boldsymbol{x}_\ell) \\
&= \underbrace{\sum_{\boldsymbol{k} \in I_{\boldsymbol{n}}} \left(b_{\boldsymbol{k}} \sum_{\ell=0}^{L-1} \alpha_\ell \, \mathrm{e}^{-2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{x}_\ell} \right) \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\boldsymbol{y}_j}}_{=: f_{\mathrm{RF}}(\boldsymbol{y}_j)} && + \underbrace{\sum_{\substack{\ell=0 \\ \|\boldsymbol{y}_j - \boldsymbol{x}_\ell\|_2 < \varepsilon_{\mathrm{I}}}}^{L-1} \alpha_\ell \, (\mathcal{K} - \mathcal{K}_{\mathrm{R}})(\boldsymbol{y}_j - \boldsymbol{x}_\ell)}_{=: f_{\mathrm{NE}}(\boldsymbol{y}_j)}.
\end{aligned} \tag{4.6}$$

In the current implementation, the kernel $\mathcal{K}\colon \mathbb{R}^d \to \mathbb{C}$ may be an arbitrary function without singularities except for the origin in the one-dimensional case and a radial function without singularities except for the origin in the multi-dimensional case, respectively. Furthermore, the target nodes $\boldsymbol{y}_j$ and source nodes $\boldsymbol{x}_\ell$ must be from the ball $\left\{\boldsymbol{x} \in \mathbb{R}^d :\ \|\boldsymbol{x}\|_2 \le 1/4 - \varepsilon_{\mathrm{B}}/2\right\}$.

The NFFT-based fast summation consists of the following (major) steps.

1. Precomputing step:
   a) node independent: calculate Fourier coefficients $b_{\boldsymbol{k}}$ by (4.5),
   b) (source and target node) dependent: precompute entries of (NFFT) matrix $\boldsymbol{B}$ for source and target nodes,
   c) source node dependent: initialize internal data structures for the efficient evaluation of the sums in $f_{\mathrm{NE}}(\boldsymbol{y}_j)$, i.e. either build a search tree for the source nodes $\boldsymbol{x}_\ell$ or sort $\boldsymbol{x}_\ell$ into boxes of edge length ε_{I} that partition the box $[-1/4+\varepsilon_{\mathrm{B}}/2,\ 1/4-\varepsilon_{\mathrm{B}}/2]^d$.
2. Calculation step (see equation (4.6)):
   a) $\mathrm{NFFT}^{\mathrm{H}}$ of the coefficients α_ℓ at the source nodes $\boldsymbol{x}_\ell$,
   b) multiplication of the resulting Fourier coefficients from sub-step 2a and the Fourier coefficients $b_{\boldsymbol{k}}$ from sub-step 1a,
   c) obtain $f_{\mathrm{RF}}(\boldsymbol{y}_j)$ by NFFT of resulting coefficients from sub-step 2b at the target nodes $\boldsymbol{y}_j$,
   d) near field correction, evaluate $f_{\mathrm{NE}}(\boldsymbol{y}_j)$ and obtain approximation $f(\boldsymbol{y}_j) \approx f_{\mathrm{RF}}(\boldsymbol{y}_j) + f_{\mathrm{NE}}(\boldsymbol{y}_j)$.

In the node independent sub-step 1a, the regularized kernel $\mathcal{K}_{\mathrm{R}}$ is evaluated for all $\boldsymbol{l} \in I_{\boldsymbol{n}}$. Since the function evaluations are independent from one another, the OpenMP pragma

```
#pragma omp parallel for default(shared) private(l)
```

was added without further modifications. Afterwards, the parallel FFTW is executed.

In sub-step 1b, either the function `nfft_precompute_lin_psi`, `nfft_precompute_psi` or `nfft_precompute_full_psi` is called, which all were already parallelized as described in section 4.2.

The initialization of the internal data structures in sub-step 1c was not parallelized due to heavy data dependencies and since this sub-step only requires a small part of the total computation time as benchmarks show.

For the sub-steps 2a and 2c, the parallel version of the nfft kernel from section 4.2 is used. The multiplication in sub-step 2b is parallelized by adding an OpenMP pragma as for sub-step 1a.

The near field correction in sub-step 2d consists of an outer for loop iterating over the target nodes $\boldsymbol{y}_j$, $y = 0, \ldots, M-1$, and a call to a helper function, which efficiently determines one of the values $f_{\mathrm{NE}}(\boldsymbol{y}_j)$ in each iteration using either the search tree or the partitioning of the box $[-1/4+\varepsilon_{\mathrm{B}}/2,\ 1/4-\varepsilon_{\mathrm{B}}/2]^d$ from sub-step 1c. As in previous sub-steps, the OpenMP pragma

```
#pragma omp parallel for default(shared) private(j)
```

was added.

Furthermore, a function call to the FFTW function `fftw_init_threads()` was added at the beginning of the `main()` function in `applications/fastsum/fastsum_test.c` as described in section 3.2.

4.5 MATLAB interface

Subsequently, the modifications to the MATLAB interface for adding the OpenMP support are described.

The current implementation of the MATLAB interface for both the nfft and nfsft kernel overwrite the memory allocation and deallocation functions `nfft_malloc` and `nfft_free` of the NFFT3 library via function hooking. When the `nfft_malloc` function is called, it calls the `nfft_mex_malloc` function provided by the MATLAB interface instead of using the FFTW memory allocation function (`fftw_malloc` for FFTW with double precision). Likewise, when the `nfft_free` function is called, it calls the `nfft_mex_free` function instead of using the FFTW memory deallocation function. The functions `nfft_mex_malloc` and `nfft_mex_free` internally call the MATLAB memory functions `mxMalloc` and `mxFree`, respectively.

However, the MATLAB memory functions `mxMalloc` and `mxFree` are not thread-safe. Calling these functions from parallelized code will likely cause MATLAB to exit abnormally. For instance, in the current implementation of the nfsft kernel (see section 4.3), several FPTs may be initialized in the `nfsft_precompute` function from OpenMP threads, where the functions `fpt_init` and `fpt_precompute` both contain many calls to the `nfft_malloc` function. Therefore, OpenMP critical sections were added to the `nfft_mex_malloc` and `nfft_mex_free` functions in `matlab/malloc.c`. Thereby, the resulting code of the `nfft_mex_malloc` function looks like:

```
void *nfft_mex_malloc(size_t n)
{
  void *p;
  #pragma omp critical (nfft_omp_matlab)
  {
    p = mxMalloc(n);

    if (!p)
      mexErrMsgTxt("Not enough memory.");

    mexMakeMemoryPersistent(p);
  }
  return p;
}
```

The remaining code of the MATLAB interface was not modified.

Furthermore, two options were added to the `configure` script, these options are described in table 4.3. For instance, the following `configure` options prepare building the multi-thread OpenMP version of the NFFT3 library and the non-parallel version of the MATLAB interface (with MATLAB installed in `/usr/shared/packages/matlabr2012a`):

```
./configure --with-matlab=/usr/shared/packages/matlabr2012a
            --enable-nfsft --enable-openmp --disable-matlab-threads
```

option	description
`--with-matlab-fftw3-libdir=DIR`	compile the MATLAB interface with the FFTW library from the directory `DIR` instead of using the FFTW library from MATLAB or the system
`--enable-matlab-threads`	compile the MATLAB interface with OpenMP support, set per default if multi-thread OpenMP version of the NFFT3 library is built (option `--enable-openmp`), `--disable-matlab-threads` allows building the non-parallel version of the MATLAB interface when the multi-thread version of the NFFT3 library is built

Table 4.3: Options added to the `configure` script concerning the MATLAB interface.

5 Benchmark results

All tests were performed on a computer named `riemann` with two Intel Xeon X5690 3.47GHz CPUs (12MB CPU cache and 6 physical CPU cores each, 12 physical CPU cores total), 144GB DDR3/PC1333 RAM and running openSUSE Linux 11.4. Computation time measurements were repeated five times and the average computation time was used.

5.1 nfft kernel

Let $\boldsymbol{N} := (N_0, \ldots, N_{d-1})^\top \in 2\mathbb{N}^d$ be the multibandlimit, $N := N_0 = N_1 = \ldots = N_{d-1}$, $\sigma > 1$ the oversampling factor, M the number of sampling nodes and m the cut-off parameter. The Fourier coefficients are chosen uniform-randomly from $[0,1] + \mathrm{i}[0,1]$ and the sampling nodes from $\mathbb{T}^d$.

For the benchmarks performed in this section, the entries of the matrix $\boldsymbol{B}$ (last step of the NFFT) and of the matrix $\boldsymbol{B}^\top$ (first step of the NFFT$^\mathsf{H}$) are not precomputed. This is denoted by "nopsi" in the plots. If the NFFT precomputation flag `PRE_PSI` was used and the precomputing time was also taken into consideration, the total NFFT/NFFT$^\mathsf{H}$ computation time would be almost identical with and without precomputation.

In this section, the speedup of the OpenMP parallelization is determined. Furthermore, the effects of sorting the sampling nodes (flag `NFFT_SORT_NODES`, see section 4.2) and of using the flag `NFFT_OMP_BLOCKWISE_ADJOINT` (cf. section 4.2) for the NFFT$^\mathsf{H}$ are determined. Thereby, "unsorted" denotes the NFFT/NFFT$^\mathsf{H}$ runs without setting the flags `NFFT_SORT_NODES` and `NFFT_OMP_BLOCKWISE_ADJOINT`, "sorted" the runs with flag `NFFT_SORT_NODES` set, and "blockwise adjoint" the runs with both flags set.

The performance gain of the parallel implementation is compared with the fastest (available) serial implementation of the NFFT and NFFT$^\mathsf{H}$, respectively. Therefore, as base time for calculating the speedup, the lowest (non-parallel) computation time from "unsorted" / "sorted" is used for the NFFT and from "unsorted" / "sorted" / "blockwise adjoint" for the NFFT$^\mathsf{H}$. For calculating the speedup at one thread, the non-parallel computation times are used, i.e. the lowest of the two / three (non-parallel) computation times is divided by the respective (non-parallel) computation time, resulting in a speedup of exactly one (at one thread) for the fastest algorithm and of not greater than one for the remaining algorithms.

The code for performing benchmarks and creating plots similar to the ones in this section can be found in `examples/nfft/nfft_benchomp.c` (performs all benchmarks and creates plots) and in `examples/nfft/nfft_benchomp_detail.c` (runs single benchmark).

5.1.1 1d

Figures 5.1 and 5.2 show the measured computation times for each step of the one-dimensional NFFT and NFFT$^\mathsf{H}$ with unsorted sampling nodes. Thereby, the computation times of the third step of the NFFT (multiplication matrix $\boldsymbol{B}$) and of the first step of the NFFT$^\mathsf{H}$ (multiplication matrix $\boldsymbol{B}^\top$) are significantly higher than the computation times of the other steps. Furthermore, the computation times of the aforementioned steps and the total computation times considerably decrease for an increasing number of computation threads until 8 threads, the computation times of the FFTW considerably decrease until 4 threads. It is important to mention that only third step of the NFFT and the first step of the NFFT$^\mathsf{H}$ depend on the number of sampling nodes M and the cut-off parameter m and each of these both steps has a computational complexity of $\mathcal{O}\left((2m+2)^d M\right)$ arithmetic operations. Therefore, the computation times of the aforementioned steps can be drastically lower or higher than shown in the plots. For instance, if the number of sampling nodes M was only 1/10 of the value in the plots, the second step of the NFFT/NFFT$^\mathsf{H}$ (FTTW) would be the most time-consuming step.

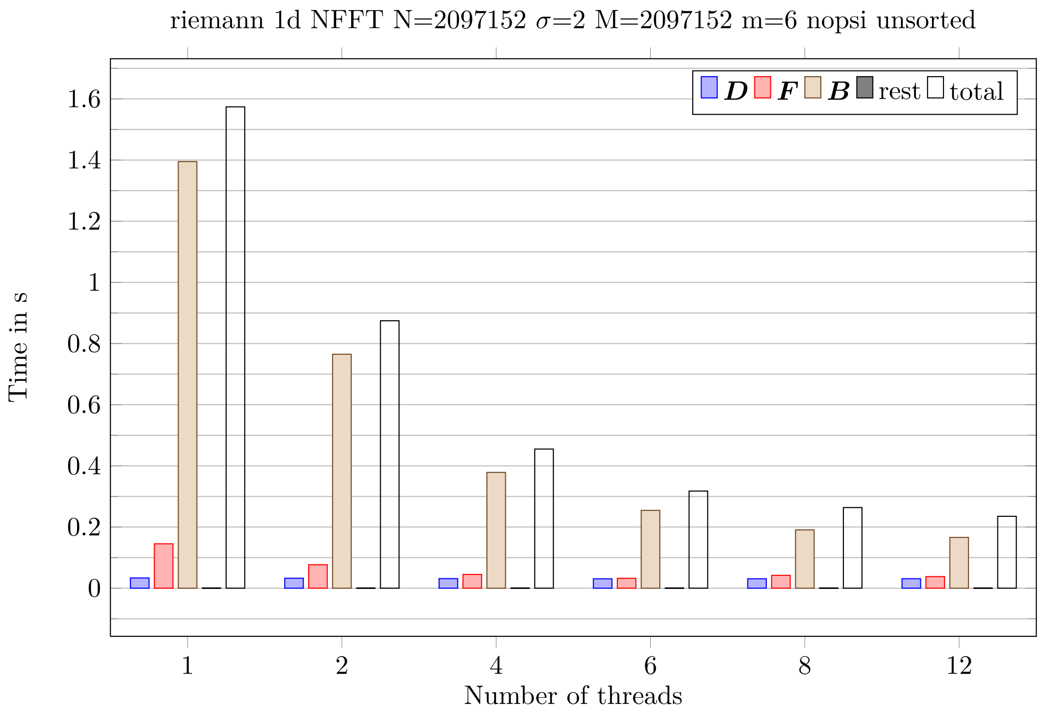

Figure 5.1: Computation times of 1d NFFT with multibandlimit $N = 2^{21} = 2097152 = |I_{\boldsymbol{N}}|$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{21} = 2097152$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for unsorted sampling nodes.

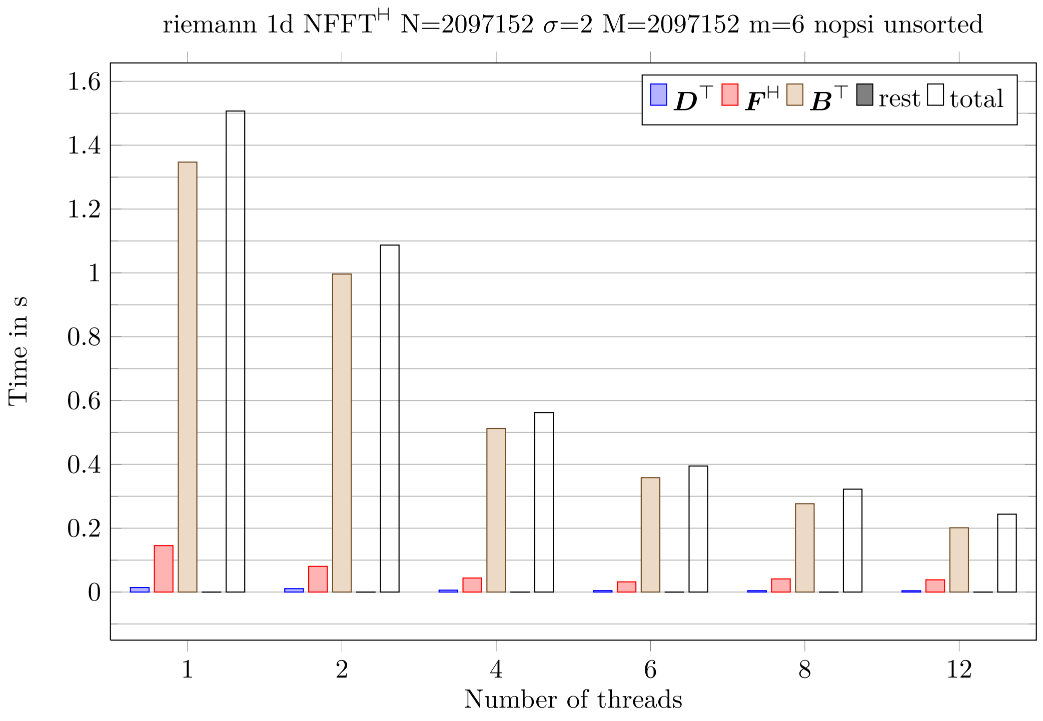

Figure 5.2: Computation times of 1d NFFT$^{\mathsf{H}}$ with multibandlimit $N = 2^{21} = 2097152 = |I_{\boldsymbol{N}}|$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{21} = 2097152$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for unsorted sampling nodes.

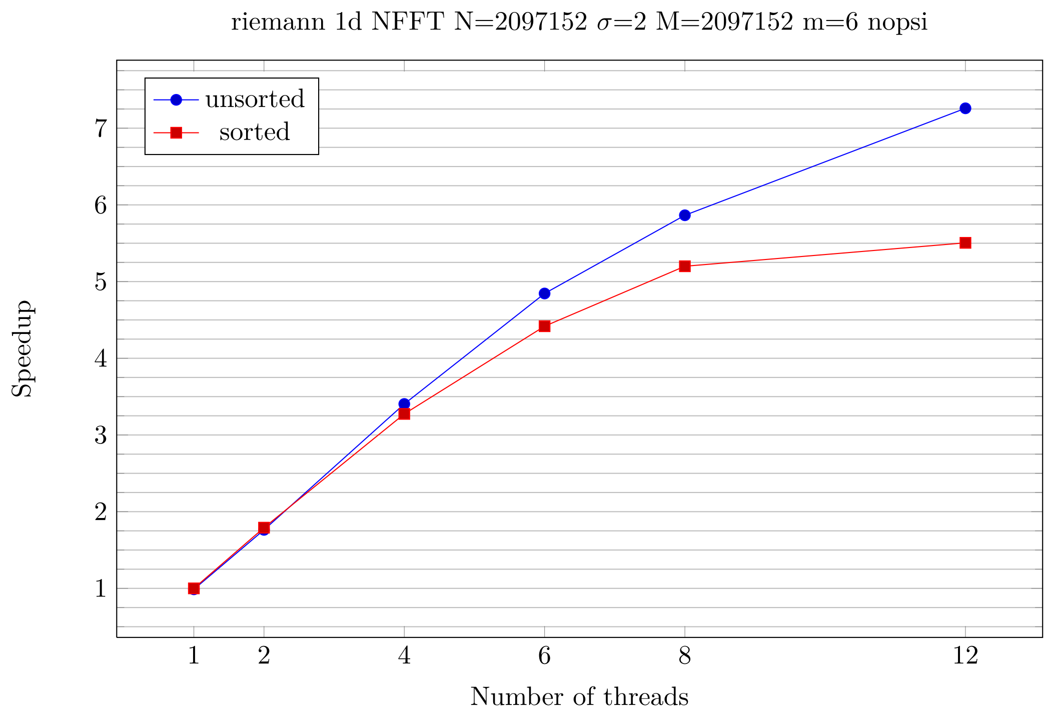

Figure 5.3: Speedup of 1d NFFT with multibandlimit $N = 2^{21} = 2097152 = |I_{\boldsymbol{N}}|$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{21} = 2097152$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for unsorted and sorted sampling nodes.

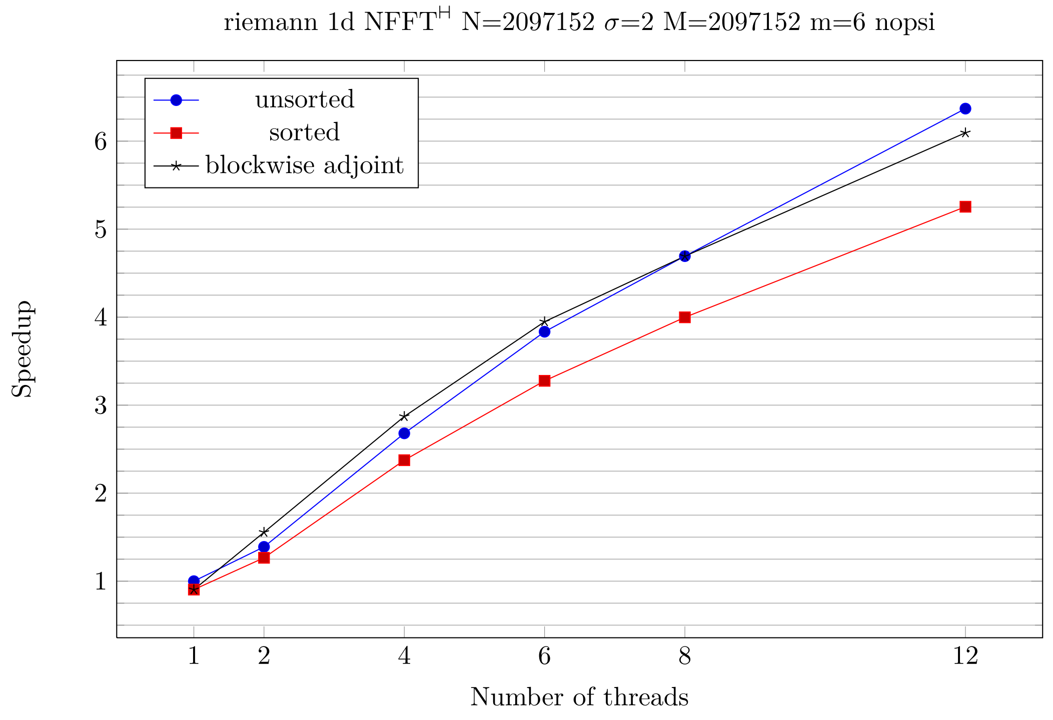

Figure 5.4: Speedup of 1d NFFT$^{\mathsf{H}}$ with multibandlimit $N = 2^{21} = 2097152 = |I_{\boldsymbol{N}}|$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{21} = 2097152$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for atomic updates with unsorted and sorted sampling nodes as well as flag `NFFT_OMP_BLOCKWISE_ADJOINT` set ("blockwise adjoint").

In addition, figure 5.3 shows the speedups for the runs of the one-dimensional NFFT, figure 5.4 for the runs of the one-dimensional NFFT$^{\mathsf{H}}$. For the chosen parameters, the results show that with 12 threads a speedup of 7.26 (efficiency 0.61) for the NFFT and of about 6.37 (efficiency 0.53) can be reached. Furthermore, not sorting the sampling nodes seems to be faster for the NFFT at all considered thread numbers and for the NFFT$^{\mathsf{H}}$ at 12 threads. The cause for this is very likely the time required sorting the nodes.

5.1.2 2d

Figure 5.5 shows the speedups for the runs of the two-dimensional NFFT, figure 5.6 for the runs of the two-dimensional NFFT$^{\mathsf{H}}$. In the two-dimensional case, sorting the sampling nodes results in a considerably higher speedup than the "unsorted" variant for the chosen parameters. Setting the flag `NFFT_OMP_BLOCKWISE_ADJOINT` (graph "blockwise adjoint") more than doubles the speedup once again for the NFFT$^{\mathsf{H}}$. The results show that with 12 threads a speedup of 7.51 (efficiency 0.63) for the "sorted" variant of the NFFT and of about 7.23 (efficiency 0.60) for the "blockwise adjoint" variant of the NFFT$^{\mathsf{H}}$ can be reached.

One important cause for the higher speedup of the "sorted" runs is that the chosen sort order can heavily reduce the cache misses in the third step of the NFFT (matrix $\boldsymbol{B}$) and in the first step of the NFFT NFFT$^{\mathsf{H}}$ (matrix $\boldsymbol{B}^\top$) when accessing the values $\boldsymbol{g}_l$, cf. section 5.1.3 for concrete numbers in the three-dimensional case. The better speedup of the "blockwise adjoint" variant of the NFFT$^{\mathsf{H}}$ is most likely due to the removal of the $(2M(2m+2)^d)$ synchronization operations (OpenMP `atomic` operations) when updating the values $\boldsymbol{g}_l$ in the first computation step (cf. section 4.2.5).

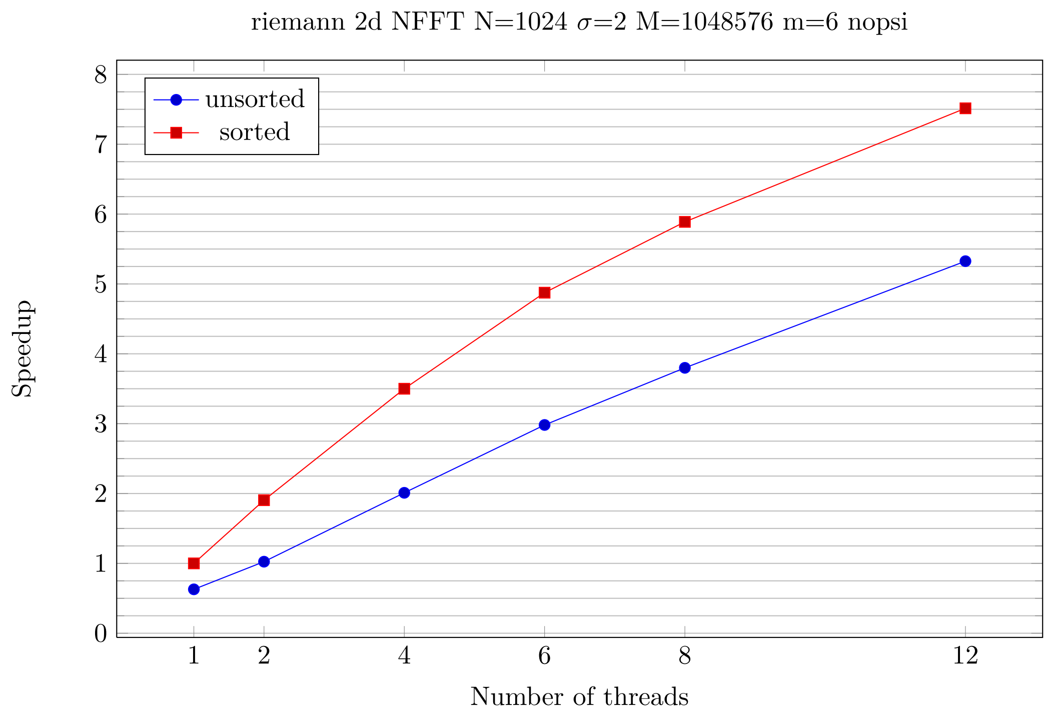

Figure 5.5: Speedup of 2d NFFT with multibandlimit $N = 2^{10} = 1024$, $|I_{\boldsymbol{N}}| = 2^{20}$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{20} = 1048576$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for unsorted and sorted sampling nodes.

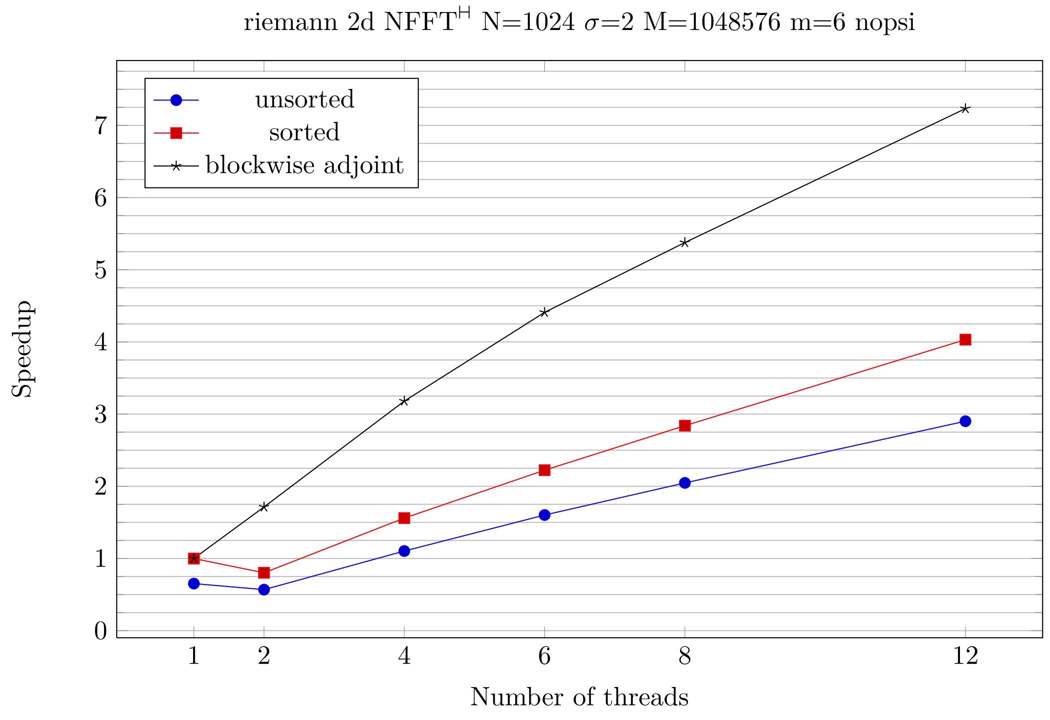

Figure 5.6: Speedup of 2d NFFT$^{\mathsf{H}}$ with multibandlimit $N = 2^{10} = 1024$, $|I_{\boldsymbol{N}}| = 2^{20}$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{20} = 1048576$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for atomic updates with unsorted and sorted sampling nodes as well as flag `NFFT_OMP_BLOCKWISE_ADJOINT` set (“blockwise adjoint”).

5.1.3 3d

Figure 5.7 shows the speedups for the runs of the three-dimensional NFFT, figure 5.8 for the runs of the three-dimensional NFFT$^{\mathsf{H}}$. As in the two-dimensional case, the “sorted” variant has a considerably higher speedup than the “unsorted” for the chosen parameters and setting the flag `NFFT_OMP_BLOCKWISE_ADJOINT` more than doubles the speedup compared to the “sorted” variant in case of the NFFT$^{\mathsf{H}}$. The results show that with 12 threads a speedup of 9.47 (efficiency 0.79) for the “sorted” variant of the NFFT and of about 7.51 (efficiency 0.63) for the “blockwise adjoint” variant of the NFFT$^{\mathsf{H}}$ can be reached.

Running the `valgrind` module `CacheGrind` on the non-parallel version of the three-dimensional NFFT showed a dramatically smaller (last-level) CPU cache miss rate of 0.1% instead of 3.3% and a total amount of $\approx$ 73 millon read misses instead of $\approx$ 1726 million. Since only the last step of the NFFT (matrix $\boldsymbol{B}$) depends on the sampling nodes and the matrix $\boldsymbol{B}$ is computed on-the-fly (“nopsi”), the most likely explaination is that the sort order of the sampling nodes heavily reduces the number of cache misses when accessing (reading) the values $\boldsymbol{g}_l$.

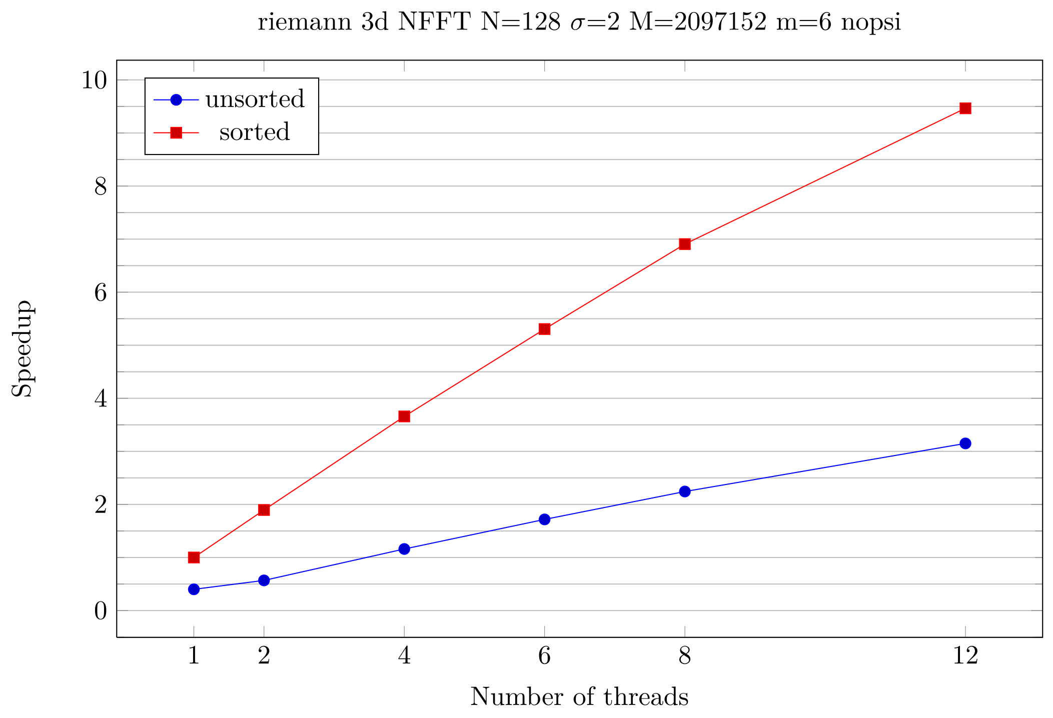

Figure 5.7: Speedup of 3d NFFT with multibandlimit $N = 2^7 = 128$, $|I_{\boldsymbol{N}}| = 2^{21}$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{21} = 2097152$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for unsorted and sorted sampling nodes.

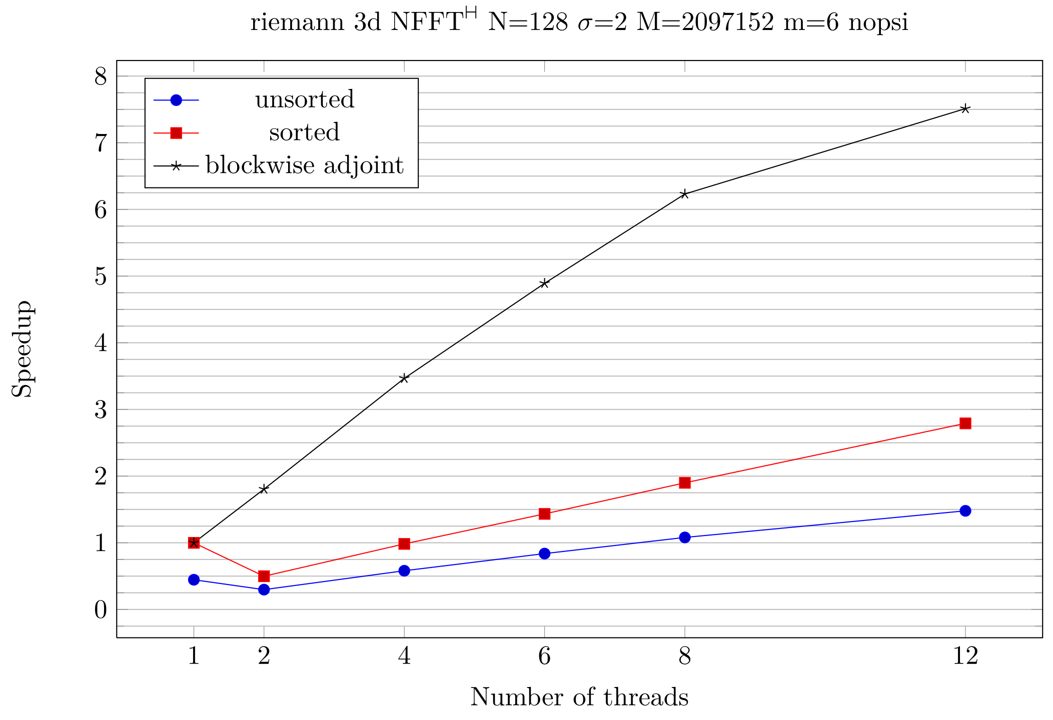

Figure 5.8: Speedup of 3d NFFT$^{\mathsf{H}}$ with multibandlimit $N = 2^7 = 128$, $|I_{\boldsymbol{N}}| = 2^{21}$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2^{21} = 2097152$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for atomic updates with unsorted and sorted sampling nodes as well as flag `NFFT_OMP_BLOCKWISE_ADJOINT` set ("blockwise adjoint").

5.2 nfsft kernel

Let N be the multibandlimit, $\sigma := 2$ the oversampling factor of the NFFT, M the number of sampling nodes and m the cut-off parameter of the NFFT The Fourier coefficients are chosen uniform-randomly from $[0, 1] + \mathrm{i}[0, 1]$ and the sampling nodes from the sphere $\mathbb{S}^2$.

For the benchmarks performed in this section, the entries of the matrix $\boldsymbol{B}$ (last step of the NFFT) and of the matrix $\boldsymbol{B}^\top$ (first step of the NFFT$^\mathsf{H}$) are not precomputed in the NFFT/NFFT$^\mathsf{H}$ step. This is denoted by "nopsi" in the plots. The NFFT is executed with the flag `NFFT_SORT_NODES` and the NFFT$^\mathsf{H}$ with additionally with the flag `NFFT_OMP_BLOCKWISE_ADJOINT` (cf. section 4.2).

In this section, the computation times of the three main steps of the NFSFT/NFSFT$^\mathsf{H}$ and the total speedup are determined. As base time for calculating the speedup and for the computation time at one thread, the computation time for the non-parallel version of the NFSFT / NFSFT$^\mathsf{H}$ is used such that the speedup for one thread is one per definition.

The code for performing benchmarks and creating plots similar to the ones in this section can be found in `examples/nfsft/nfsft_benchomp.c` (performs all benchmarks and creates plots) and in `examples/nfsft/nfsft_benchomp_detail.c` (runs single benchmark).

In the histogram of the NFSFT computation times in figure 5.10, the FPT and NFFT show approximately the same computation time whereas the step `c2e` requires an insignificant time. Furthermore, the total computation time distinctly reduces at increasing number of threads until 8 threads. The histogram of the NFSFT$^\mathsf{H}$ in figure 5.11 looks similar despite the higher computation times of the FPT$^\top$.

The corresponding speedup plot of the total computation time in figure 5.9 shows a nearly identical speedup until up to 4 threads (≈ 3.5 at 4 threads) for the NFSFT and NFSFT$^\mathsf{H}$. At 6 threads and above, the speedup of the NFSFT$^\mathsf{H}$ is about 0.5 less than the speedup of the NFSFT. The measured speedup at 12 threads is ≈ 6.5 for the NFSFT and ≈ 5.9 for the NFSFT$^\mathsf{H}$.

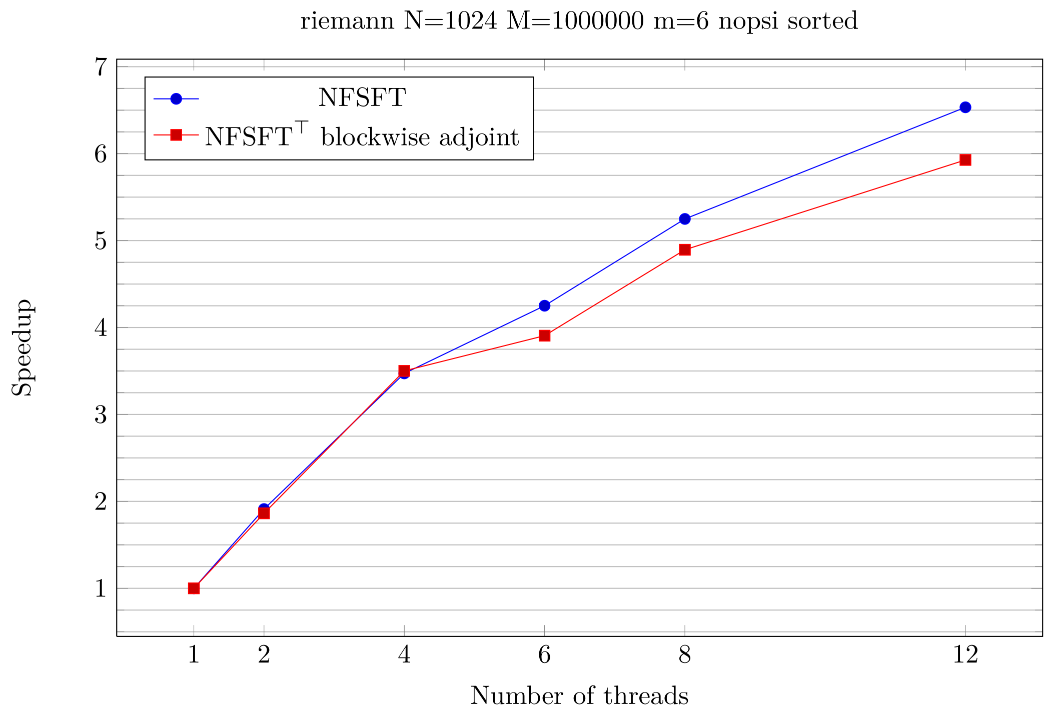

Figure 5.9: Speedup of the NFSFT and NFSFT$^\mathsf{H}$ with multibandlimit $N = 128$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 2097152$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for sorted sampling nodes (NFSFT) and for flag `NFFT_OMP_BLOCKWISE_ADJOINT` (NFSFT$^\mathsf{H}$) set.

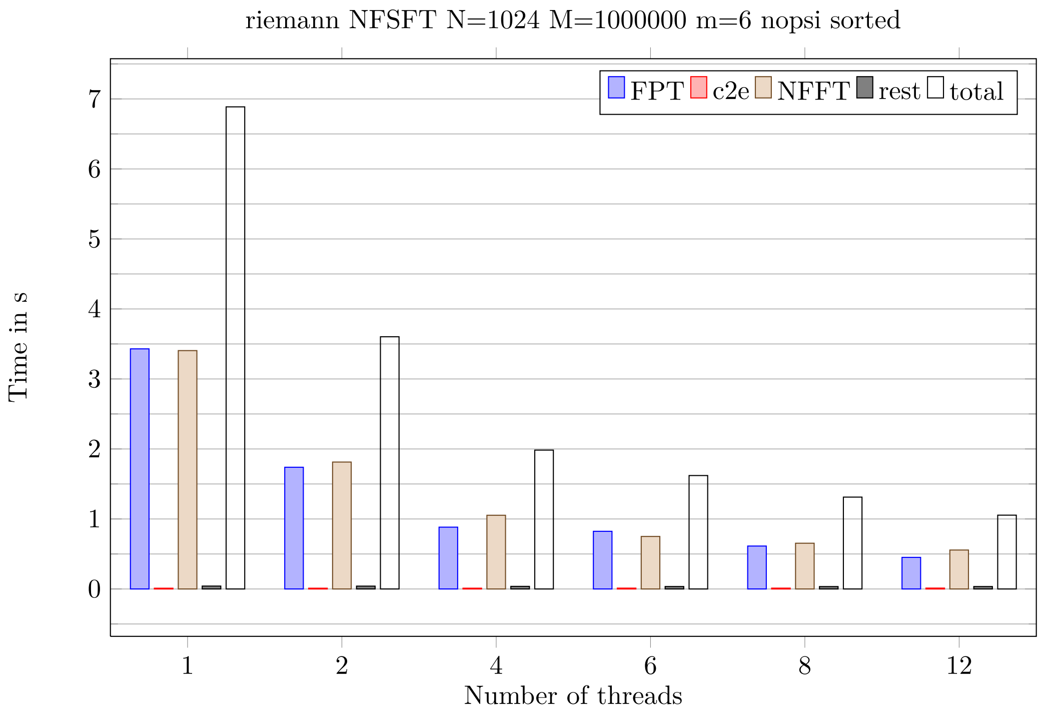

Figure 5.10: Computation times of the NFSFT with multibandlimit $N = 1024$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 1000000$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for sorted sampling nodes.

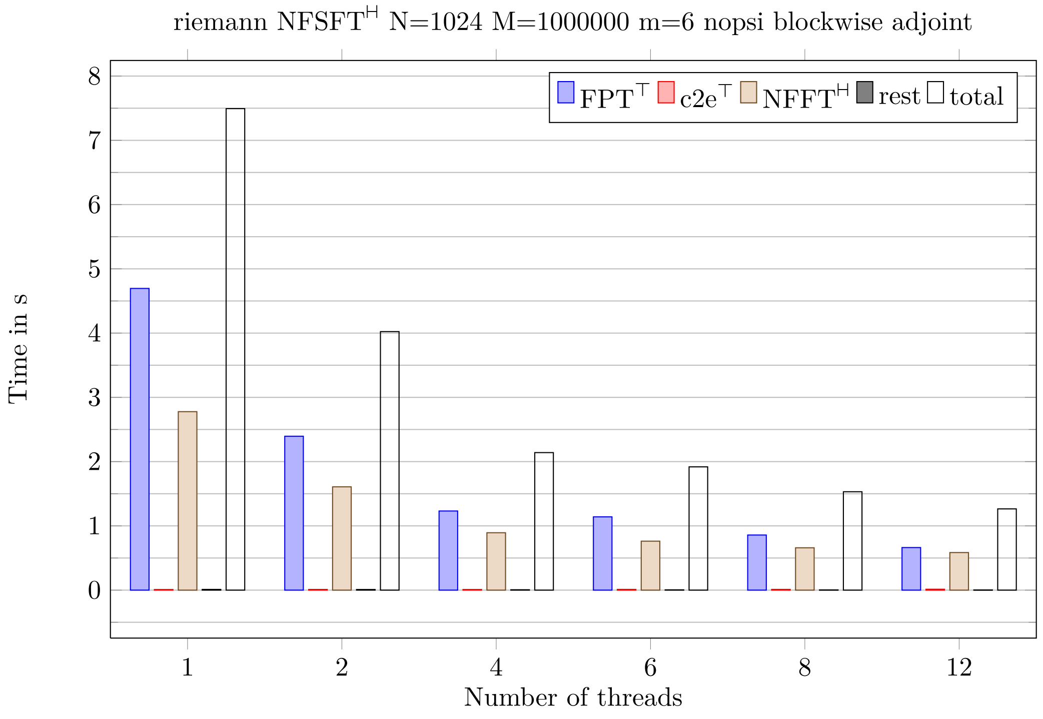

Figure 5.11: Computation times of the NFSFT^H with multibandlimit $N = 1024$, oversampling factor $\sigma = 2$, number of sampling nodes $M = 1000000$, cut-off parameter $m = 6$ and without precomputation of matrix $\boldsymbol{B}$ for flag `NFFT_OMP_BLOCKWISE_ADJOINT` (NFSFT^H) set.

5.3 NFFT-based fast summation (fastsum)

Since the pre-computation sub-step 1a of the NFFT-based fast summation (see section 4.4) does not depend on the source nodes $\boldsymbol{x}_\ell$, coefficients α_ℓ or target nodes $\boldsymbol{y}_j$, the corresponding computation times were ignored for the speedup and total computation time of the NFFT-based fast summation.

The benchmark was performed for the three dimensional case and the kernel function $\mathcal{K}(\boldsymbol{x}) = 1/\|\boldsymbol{x}\|_2$. Thereby, $L = 100000$ uniformly distributed random source nodes $\boldsymbol{x}_\ell$ and $M = 100000$ uniformly distributed random target nodes $\boldsymbol{y}_j$ were used. The fastsum and NFFT parameters were chosen in such a manner that the maximal relative error for the results of the fast summation (compared to the direct summation) is less than 10^{-6} and that the computation time for the far field (sub-steps 2a-2c) is almost equal to the one for the near field (sub-step 2d), resulting in a NFFT multibandlimit $N = 128$, a NFFT window cut-off parameter $m = 4$, fastsum smoothness parameter p and size of regularization region $\varepsilon_{\mathrm{I}} = \varepsilon_{\mathrm{B}} = 4/128$.

Figure 5.12 shows the speedup values determined for up to 12 threads. At 2 threads, the speedup is ≈ 1.87 and the efficiency is about 0.935. Then, the growth of the speedup decreases for 4 and 6 threads, where the speedup is about 3.12 and 4.13, respectively. The corresponding efficiency is about 0.78 at 4 threads and 0.689 at 6 threads. The speedup at 12 threads is ≈ 6.03 and the efficiency is ≈ 0.503.

In figure 5.13, the corresponding (total) computation times are shown as well as the computation times for the sub-steps of the fast summation. As mentioned before, the total computation time does not include the node-independent pre-computations of sub-step 1a. Thereby, the times for the far field and near field computation are almost equal to one another at one thread and two threads. At 4 threads and above, the far field computation times only slightly decrease with increasing number of threads, whereas the near field computation times show an excellent decrease. Especially at 8 and 12 threads, the total computation time is dominated by the far field. The remaining node-dependent pre-computing sub-steps only require a small fraction of the total computation time in the considered cases.

The computation times for sub-step 2a ($\mathrm{NFFT}^{\mathsf{H}}$) and sub-step 2c (NFFT) were analyzed in detail for the example of this section, the results are shown in figure 5.14 and 5.15, respectively. In contrast to the results of section 5.1.3, the $\mathrm{NFFT}^{\mathsf{H}}$ and NFFT do not parallelize well beyond 4 threads in the current example. For the $\mathrm{NFFT}^{\mathsf{H}}$, the computation time of the first computation step (matrix $\boldsymbol{B}^\top$) only slightly decreases at 4 threads and above when compared to the time at 2 threads. Furthermore, the computation times for the second step of the $\mathrm{NFFT}^{\mathsf{H}}$ and NFFT (Fourier matrix $\boldsymbol{F}$) show almost no decrease for the cases with 8 and 12 threads compared to 6 threads.

These problems have not been observed in section 5.1.3, since both the number of sampling nodes and the cut-off parameter m were chosen larger. However, if the number of source and target nodes was increased for the fast summation, for instance by ≈ 21 to $M = L = |I_{\boldsymbol{N}}|$, then the near field computation times would increase drastically by a factor of $\approx 21^3$. Furthermore, an increase of the cut-off parameter m would only cause higher computation times without achieving a better approximation error unless other parameters (e.g. the multibandlimit) were increased.

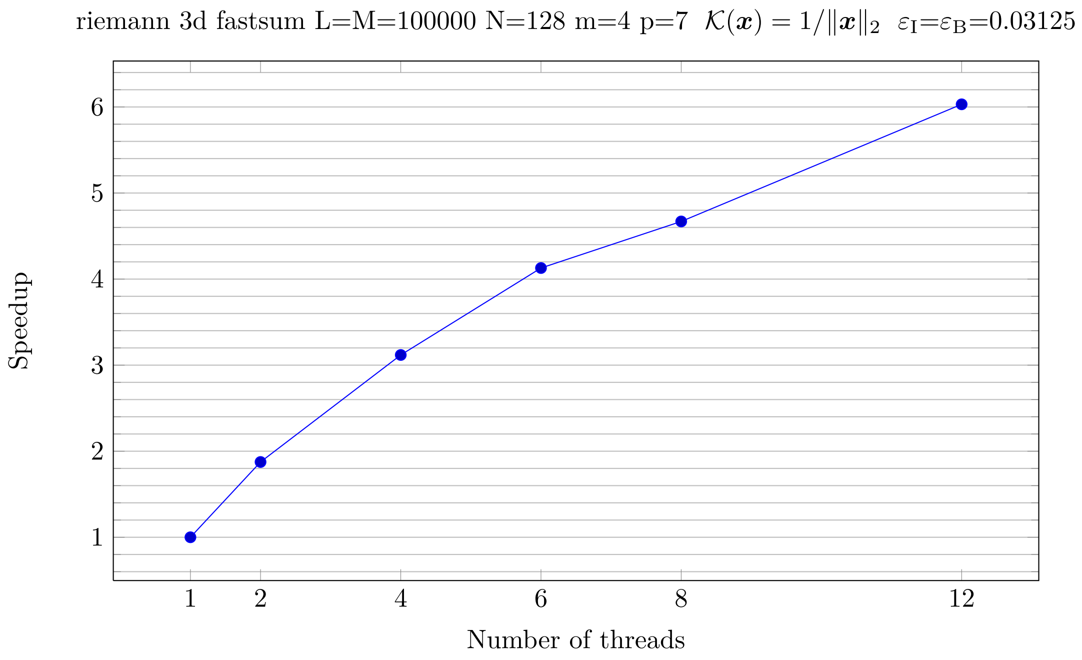

Figure 5.12: Speedup of 3d NFFT-based fast summation (fastsum) with kernel $\mathcal{K}(\boldsymbol{x}) = 1/\|\boldsymbol{x}\|_2$ and oversampling factor $\sigma = 2$ for sorted sampling nodes (NFFT) and for flag `NFFT_OMP_BLOCKWISE_ADJOINT` (NFFT$^{\mathsf{H}}$) set. Computation time for node-independent pre-computation (sub-step 1a) was ignored.

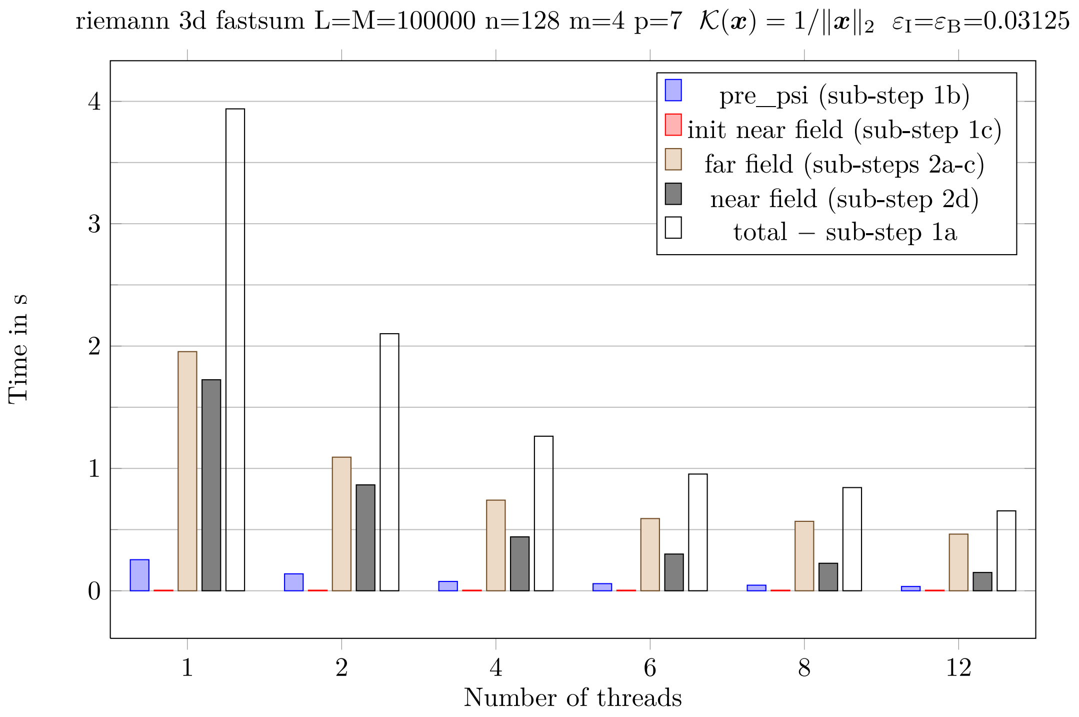

Figure 5.13: Computation times of 3d NFFT-based fast summation (fastsum) with kernel $\mathcal{K}(\boldsymbol{x}) = 1/\|\boldsymbol{x}\|_2$ and NFFT oversampling factor $\sigma = 2$ for sorted sampling nodes (NFFT) and for flag `NFFT_OMP_BLOCKWISE_ADJOINT` (NFFT$^{\mathsf{H}}$) set. Computation time for node-independent pre-computation (sub-step 1a) was ignored.

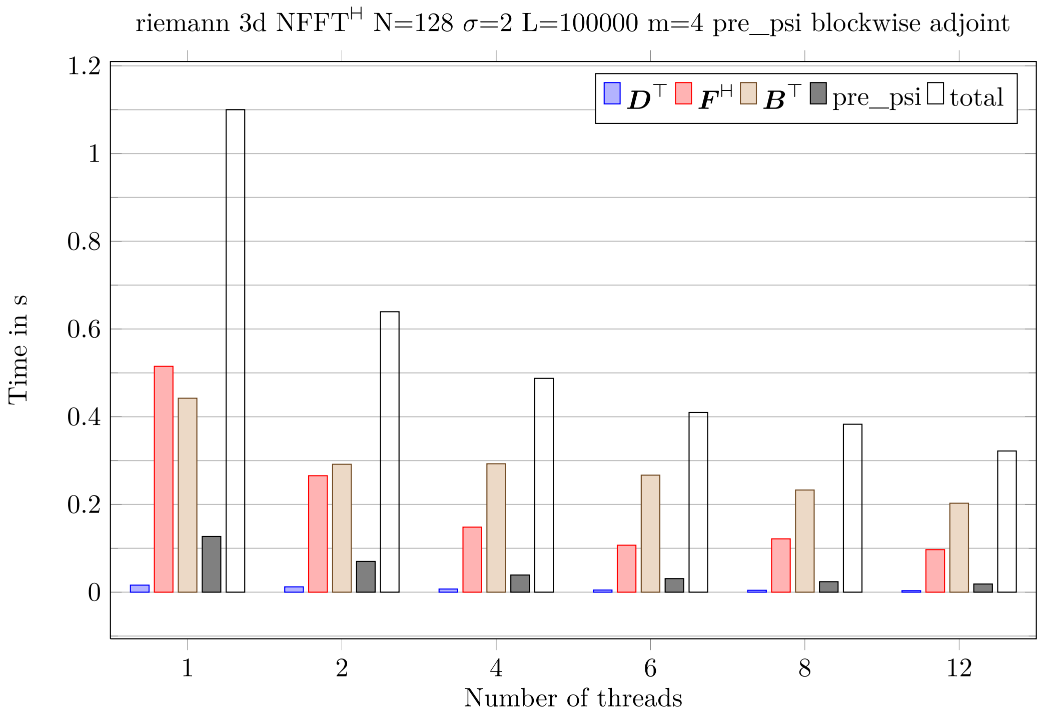

Figure 5.14: Computation times of 3d NFFT$^{\mathsf{H}}$ with multibandlimit $N = 2^7 = 128$, $|I_{\boldsymbol{N}}| = 2^{21}$, oversampling factor $\sigma = 2$, number of (source) nodes $L = 100000$, cut-off parameter $m = 4$ and with precomputation of matrix $\boldsymbol{B}$ ("pre_psi") for flag `NFFT_OMP_BLOCKWISE_ADJOINT` set.

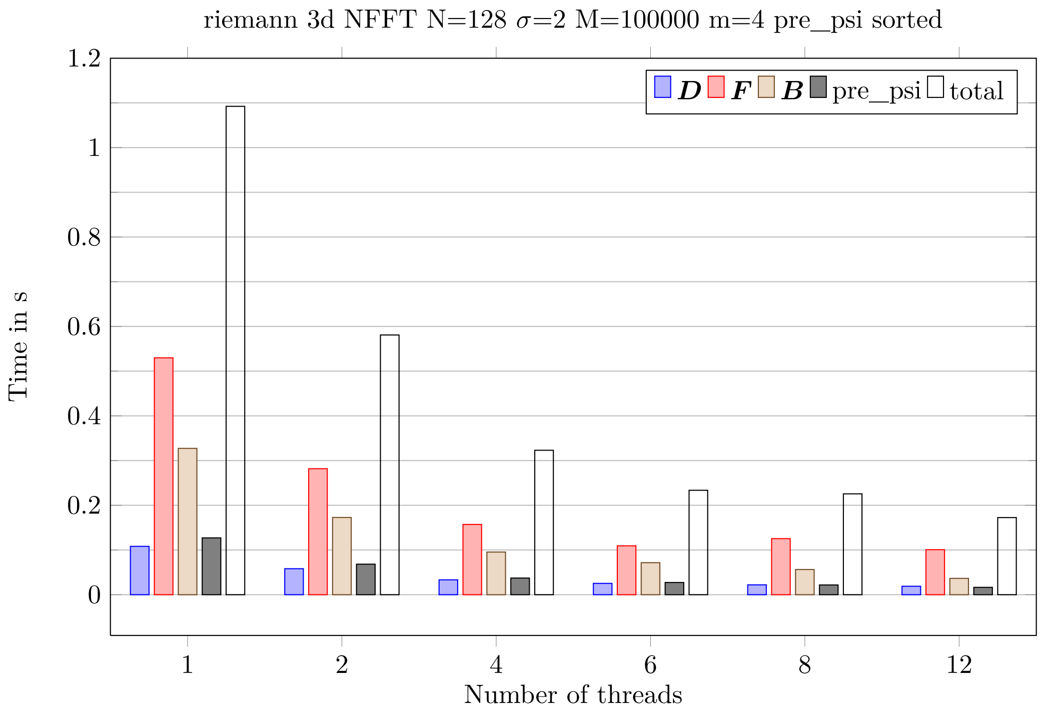

Figure 5.15: Computation times of 3d NFFT with multibandlimit $N = 2^7 = 128$, $|I_{\boldsymbol{N}}| = 2^{21}$, oversampling factor $\sigma = 2$, number of (target) nodes $M = 100000$, cut-off parameter $m = 4$ and with precomputation of matrix $\boldsymbol{B}$ ("pre_psi") for sorted sampling nodes.

Bibliography

[1] M. Fenn: *Fast Fourier transform at nonequispaced nodes and applications.* Dissertation, Universität Mannheim, 2006.

[2] M. Frigo and S.G. Johnson: *FFTW, C subroutine library.* `http://www.fftw.org`, 2009. http://www.fftw.org.

[3] J. Keiner, S. Kunis, and D. Potts: *NFFT 3.0, C subroutine library.* `http://www.tu-chemnitz.de/~potts/nfft`. http://www.tu-chemnitz.de/~potts/nfft.

[4] J. Keiner, S. Kunis, and D. Potts: *Using NFFT3 - a software library for various nonequispaced fast Fourier transforms.* ACM Trans. Math. Software, 36:Article 19, 1 – 30, 2009.

[5] D.E. Knuth: *The art of computer programming, volume 3: (2nd ed.) sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998, ISBN 0-201-89685-0.

[6] S. Kunis and D. Potts: *Fast spherical Fourier algorithms.* J. Comput. Appl. Math., 161:75 – 98, 2003.

[7] OpenMP Architecture Review Board: *OpenMP C and C++ Application Program Interface. Specification.* `http://www.openmp.org/mp-documents/cspec20.pdf`, 2002. http://www.openmp.org/mp-documents/cspec20.pdf.

[8] D. Potts and G. Steidl: *Fast summation at nonequispaced knots by NFFTs.* SIAM J. Sci. Comput., 24:2013 – 2037, 2003.

[9] D. Potts, G. Steidl, and M. Tasche: *Fast and stable algorithms for discrete spherical Fourier transforms.* Linear Algebra Appl., 275/276:433 – 450, 1998.