Parallelization of GSL: Performance of Case Studies*

José Aliaga[1], Francisco Almeida[2], José M. Badía[1], Sergio Barrachina[1], Vicente Blanco[2], María Castillo[1], U. Dorta[2], Rafael Mayo[1], Enrique S. Quintana[1], Gregorio Quintana[1], Casiano Rodríguez[2], and Francisco de Sande[2]

[1] Depto. de Ingeniería y Ciencia de Computadores, Univ. Jaume I, 12.071–Castellón, Spain
{aliaga,badia,barrachi,castillo, mayo,quintana,gquintan}@icc.uji.es

[2] Depto. de Estadística, Investigación Operativa y Computación, Univ. de La Laguna 38.271–La Laguna, Spain
{falmeida,vblanco,casiano,fsande}@ull.es

Abstract. In this paper we explore the parallelization of the scientific library from GNU both on shared-memory and distributed-memory architectures. A pair of classical operations, arising in sparse linear algebra and discrete mathematics, allow us to identify the major challenges involved in this task, and to analyze the performance, benefits, and drawbacks of two different possible parallelization approaches.

1 Introduction

The GNU Scientific Library (GSL) [4] is a collection of hundreds of routines for numerical scientific computations written in ANSI C, which includes codes for complex arithmetic, matrices and vectors, linear algebra, integration, statistics, and optimization, among others. The reason GSL was never parallelized seems to be the lack of a globally accepted standard for developing parallel applications. We believe that with the introduction of OpenMP and MPI the situation has changed substantially. OpenMP has become a standard *de facto* for exploiting parallelism using *threads*, while MPI is nowadays accepted as the standard interface for developing parallel applications following the message-passing programming model.

In this paper we investigate the parallelization of a specific part of GSL using OpenMP and MPI and their respective performances on shared-memory multiprocessors (SMPs) and distributed-memory multiprocessors (or multicomputers). In Section 2 we give a general overview of our parallel integrated version of GSL. Two classical numerical computations, arising in sparse linear algebra and discrete mathematics, are employed to explore the challenges involved in this task. In Section 3 we elaborate our first case study, describing a parallelization of the sparse matrix-vector product addressed to SMPs. In Section 4 the analysis is repeated for a parallel routine that computes the Fast Fourier Transform (FFT) targeted in this case for multicomputers. Finally, concluding remarks follow in Section 5.

* This work has been partially supported by the EC (FEDER) and the Spanish MCyT (Plan Nacional de I+D+I, TIC2002-04498-C05-05 and TIC2002-04400-C03).

2 Parallel GSL

Our general goal is to develop parallel versions of GSL using MPI and OpenMP which are portable to several parallel architectures, including distributed and shared-memory multiprocessors, hybrid systems, and clusters of heterogeneous nodes. Besides, we want to reach two different classes of users: a programmer with no experience in parallel programming, who will be denoted as user A, and a second programmer, or user B, that regularly utilizes MPI or OpenMP. As additional objectives, the routines included in our library should execute efficiently on the target parallel architecture and, equally important, the library should appear to user A as a collection of traditional serial routines. We believe our approach to be different to some other existing parallel scientific libraries (see, e.g., *http://www.netlib.org*) in that our library targets multiple classes of architectures while maintaining the sequential interface of the routines.

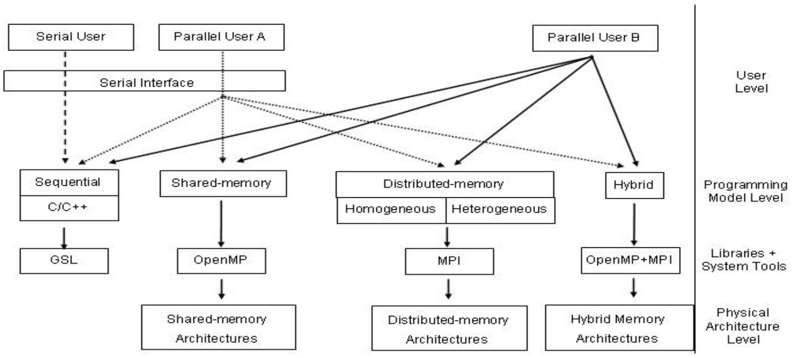


Fig. 1. Software architecture of the parallel version of GSL

Our library has been designed as a multilevel software architecture; see Fig. 1. The *User Level* (the top level) provides a sequential interface that hides the parallelism to user A and supplies the services through C/C++ functions according to the prototypes specified by the sequential GSL interface.

The *Programming Model Level* provides a different instantiation of the GSL library for each one of the computational models: sequential, distributed-memory, shared-memory, and hybrid. The semantics of the functions in the Programming Model Level are those of the parallel case so that user B can invoke them directly from her own parallel programs. The Programming Model Level implements the services for the upper level using standard libraries and parallelizing tools such as (the sequential) GSL, MPI, and OpenMP.

In the distributed-memory (or message-passing) programming model we view the parallel application as being executed by p *peer* processes, P_0, P_1,...,P_{p-1}, where the same parallel code is executed by all processes on different data.

In the *Physical Architecture Level* the design includes shared-memory platforms, distributed-memory architectures, and hybrid and heterogeneous systems.

For further details of the functionality and interfaces, see [1].

3 Case Study I: Sparse Matrix-Vector Product

3.1 Operation

Sparse matrices arise in a vast amount of areas. Surprisingly enough, GSL does not include routines for sparse linear algebra, most likely due to the lack of an standardized interface definition, which was only developed very recently [3].

Here we explore the parallelization of the *sparse matrix-vector product*, denoted as USMV in the Level-2 sparse BLAS [3]. In this operation an $m \times n$ sparse matrix A, with nz nonzero elements, is multiplied by a vector x with n elements. The result, scaled by a value α, is used to update a vector y of order m as $y \leftarrow y + \alpha \cdot A \cdot x$.

The implementation, parallelization, and performance of the USMV operation strongly depends on the storage scheme employed for the sparse matrix. In the coordinate format, two integer arrays of length nz hold the row and column indices of the nonzero elements of the matrix, while a third array, of the same dimension, holds the values. In the columnwise variant of this format, the values are listed by columns. The rowwise Harwell-Boeing scheme also employs a pair of arrays of length nz for the column indices and the values. A third array, of length $m+1$, determines then the starting/ending indices for each one of the rows of the matrix. Figure 2 illustrates the use of these two storage schemes by means of a simple example.

A 3x4 sparse matrix

	0	1	2	3
0	1.0	2.0		
1				4.0
2		3.0		

Columnwise coordinate

	0	1	2	3
rows	0	0	2	1
cols.	0	1	1	3
values	1.0	2.0	3.0	4.0

Rowwise Harwell–Boeing

	0	1	2	3
rows	0	2	3	4
cols.	0	1	3	1
values	1.0	2.0	4.0	3.0

Fig. 2. Storage schemes for a 3×4 sparse matrix with $nz = 4$ nonzero elements

The USMV operation is usually implemented as a series of *saxpy* (scalar α times x plus y) operations or *dot products* depending respectively on whether the matrix is stored columnwise or rowwise. In particular, in the columnwise implementation of the matrix-vector product, for each column of A, denoted as $A(:,j)$, a saxpy operation is performed to update vector y as $y \leftarrow y + A(:,j) \cdot (\alpha * x[j])$. In the rowwise case, the matrix-vector product is computed as a series of dot products, involving a row of matrix A, denoted as $A(i,:)$, and x. The result is used to update a single element of y as $y[i] \leftarrow y[i] + \alpha \cdot (A(i,:) \cdot x)$. Codes for the columnwise/rowwise implementation of the sparse matrix-vector product are given in Fig. 3. Minor modifications are introduced in those codes to facilitate the exposition of the parallel implementations using OpenMP.

3.2 Parallel Implementation Using OpenMP

Parallelizing a sequential code using OpenMP is, in most cases, straight-forward. One only needs to evaluate possible data dependencies and then place the appropriate directives in loops/regions of the code that indicate the OpenMP environment how to extract

```
indval=0;
for(j=0;j<n;j++){
  nzj=nz_in_column(j);
  temp=alpha*x[j];
  for(i=0;i<nzj;i++){
    k=rows[indval+i];
    y[k]+=values[indval+i]
    *temp;
  }
  indval+=nzj;
}
```

```
for(i=0;i<m;i++){
  indval=rows[i];
  temp=0.0;
  for(j=rows[i];j<rows[i+1];
    j++){
    k=cols[indval];
    temp+=values[indval]
    *x[k];
    indval++;
  }
  y[i]=y[i]+alpha*temp;
}
```

Fig. 3. Implementation of the USMV operation for the columnwise coordinate format (left) and the rowwise Harwell-Boeing format (right). Routine `nz_in_column` returns the number of nonzero elements in a column

parallelism from the corresponding parts using threads. As a general rule, in codes composed of nested loops it is better to parallelize the outermost loop as that distributes a larger part computational load among the threads.

Consider now the columnwise coordinate implementation of USMV (left of Fig. 3). Parallelizing the outer loop in this code implies that two or more threads could be concurrently updating the same element of vector y, which is a well-known case for a race condition. A solution to this problem is to guarantee exclusive access to the elements of y; however, synchronization mechanisms for mutual exclusion usually result in large overheads and performance loss. Therefore, we decide to only parallelize the inner loop in this case so that the outer loop is executed by a single *master* thread, while multiple threads will cooperate during the computation of each saxpy operation performed in the inner loop. In order to do so, we need to introduce the OpenMP directive:

```
#pragma omp parallel for private (i, k)
```

just above the inner loop. Given that each iteration of the inner loop requires the same amount of computations, a static scheduling of the threads achieves a good distribution of the computational load for this implementation. The parallelization of the inner loop, though simple, is not completely satisfactory. In general, the number of nonzero entries per column is small, while the number of columns of the matrix is much larger. Since the threads need to be synchronized every time execution of the inner loop is terminated (that is, once per column), a large overhead is again likely to arise in this approach.

The parallelization of the rowwise implementation (right of Fig. 3) using multiple threads is much easier. As each iteration of the outer loop is independent, we can parallelize this loop by introducing the directive:

```
#pragma omp parallel for private (indval, temp, j, k)
```

just before the outer loop. A certain amount of dot products are thus computed by each thread and the threads only need to be synchronized once, on termination of the outer loop. A dynamic scheduling of the threads provides a better computational load balancing in this case as usually the nonzero entries of the matrix are distributed unevenly among the rows.

Table 1. Time (in sec.) of the parallel implementations for the USMV operation

Density	Columnwise coordinate			Rowwise Harwell-Boeing		
	1 Thread	2 Threads	4 Threads	1 Thread	2 Threads	4 Threads
0.01	0.031	1.961	0.485	0.029	0.021	0.012
0.1	0.292	1.676	0.608	0.207	0.162	0.097
1.0	2.855	2.389	1.390	1.976	1.560	0.917

3.3 Experimental Results

All the experiments in this subsection were obtained on a 4-way SMP UMA platform consisting of 4 Intel Pentium Xeon@700MHz processors with 2.5 Gbytes of RAM and a 1 Mbyte L3 cache. We employed the Omni 1.6 portable implementation of OpenMP for SMPs (`http://phase.hpcc.jp/Omni/`).

We report the performance of the parallel implementations of the USMV operation for a sparse matrix with $m = n = 50,000$ rows/columns and a rate of nonzero elements (density) ranging from 0.01$\%$ to 1$\%$. The sparse matrices are generated so that the elements are randomly distributed among the rows/columns of the matrix resulting in a similar number of nonzero elements per row/column. The number of (floating-point arithmetic) operations of the sparse matrix-vector product is proportional to the square of the number of nonzero entries of the matrix, and therefore the computational cost of this problem can be considered as moderate. In other words, one should not expect large speed-ups unless the matrix becomes "less sparse". However, sparse matrices with a density rate larger than 1$\%$ are rare.

Table 1 shows the execution times of the sequential code (columns labeled as 1 thread) and the parallel columnwise/rowwise codes using 2 and 4 threads. Comparing only the sequential codes, the columnwise implementation obtains larger execution times. This is usual as current architectures, where the memory is structured hierarchically in multiple levels, benefit more from an operation such as the dot product than saxpys. One could then argue against the use of the columnwise code. However, notice that once the storage scheme of the matrix is fixed as rowwise, the computation of $y \leftarrow y + \alpha \cdot A^T \cdot x$, an operation probably as frequent as the non-transposed matrix-vector product, requires accessing the elements of the matrix as in the columnwise code.

Now, let us consider the parallel performances of both variants. The columnwise implementation offers poor parallel results until the density of nonzero elements reaches 1$\%$. This behaviour was already predicted in the previous subsection: As the threads need to be synchronized once per column, when the number of nonzero entries per column is small compared with the number of columns of the matrix, the overhead imposed by the synchronization degrades the performance of the algorithm. Only when the density rate is 1$\%$ the parallel implementation outperforms the serial code, with speed-ups of 1.19 and 2.05 for 2 and 4 threads, respectively.

For the rowwise implementation, the experimental results show maintained speed-ups around 1.3 and 2.4 using 2 and 4 threads, respectively, for all density rates. Further experimentation is needed here to evaluate whether a different implementation of OpenMP, or even a different platform, would offer better performances.

4 Case Study II: FFT

4.1 Operation

The Fast Fourier Transform (FFT) plays an important role in many scientific and engineering applications arising, e.g., in computational physics, digital signal processing, image processing, and weather simulation and forecast.

We analyze hereafter the parallelization of the FFT routines in GSL. The case when the number of processors is a power of two has been extensively analyzed, and we employ here the version in [5]. The FFT algorithms can be generalized for the case when neither the number of processors nor the problem size, n, are a power of two. However, in general, this introduces a non-negligible communication overhead and results in a considerable load imbalance. We follow the parallel mixed-radix FFT algorithm presented in [2], which is a variant of Temperton's FFT [6]. The algorithm presents a low communication overhead, has good load balance properties, is independent of the number of processors, and poses no restrictions on the number of processors nor the size of the input vector.

We start by introducing some preliminary notation needed for the description of the FFT algorithm [2]:

- Rows and columns of matrices are indexed starting from 0.
- Element (j, k) of matrix A is stated to as $\mid A \mid (j, k)$.
- The Kronecker product of matrices A and B is denoted as $A \otimes B = (a_{ij}B)$.
- The permutation matrix P_q^p of order pq is defined as

$$\mid P_q^p \mid (j,k) = \begin{cases} 1 & \text{if } j = rp + s \text{ and } k = sq + r, \\ 0 & \text{otherwise.} \end{cases}$$

- The diagonal matrix D_q^p of order pq is defined as

$$\mid D_q^p \mid (j,k) = \begin{cases} w^{sm} & \text{if } j = k = sq + m, \\ 0 & \text{otherwise,} \end{cases}$$

  where $w = \exp(2\pi\imath/pq)$, $\imath = \sqrt{-1}$.
- I_m states for the square identity matrix of order m.

By definition, the Discrete Fourier Transform (DFT) of $z = (z_0, ..., z_{n-1})^T \in \boldsymbol{C}^n$ is given by

$$x_j = \sum_{k=0}^{n-1} z_k \exp(2\pi jk\imath/n), \quad 0 \le j \le n, \quad x \in \boldsymbol{C}^n. \tag{4.1}$$

Alternatively,

$$x = W_n z, \quad W_n(j,k) = w^{jk}, \quad w = \exp(2\pi\iota/n). \tag{4.2}$$

Here, W_n is known as the DFT matrix of order n. The idea behind the FFT algorithm is to factorize W_n so that the multiplication by W_n is broken up into $O(n \log n)$ multiplications involving smaller matrices (each one of a small constant size). Temperton presented in [6] several variants of the FFT algorithm by rearranging the terms as different factorized multiplications of the form $W_n z$. In particular, for $n = f_1 f_2 \cdots f_{k-1} f_k$,

$$W_n = T_k\, T_{k-1} \cdots T_2\, T_1\, P_1\, P_2 \cdots P_{k-1}\, P_k, \tag{4.3}$$

with

$$T_i = (I_{m_i} \otimes W_{f_i} \otimes I_{l_i})\,(I_{m_i} \otimes D^{f_i}_{l_i}), \quad P_i = (I_{m_i} \otimes P^{l_i}_{f_i}),$$

$l_i = 1, l_{i+1} = f_i l_i, m_i = n/l_{i+1}$, and $1 \le i \le k$. Notice that all permutation matrices in (4.3) are shifted to the right. From this formulation we arrive at the following sequential algorithm for the FFT, with complexity $O(n \sum f_i)$:

```
Factorize(n, k);  // Compute f[0], ..., f[k-1]
for (i = k-1; i >= 0; i--)
  z = P[i] * z;
for (i = 0; i < k; i++)
   z = T[i] * z;  // m[i]*l[i] sub-DFTs of size n
                  // have to be solved
```

While the first loop computes a permutation of the input array, the parallelization effort is focused in the second loop.

4.2 Parallel Implementation Using MPI

The parallelization of the above algorithm has been also presented in [2] and we describe it with the code below. The algorithm is based on a particular layout of the data to reduce the communication overhead with a proper load balance. At the beginning of each iteration, processor p_j receives just the data needed for the computation during this iteration, according to a Rolled_Break_Cyclic distribution (to be described later; see fig. 4). Only f_i communication steps among processors $p_j - 1$ and $p_j + 1$ are involved. After the computation of the local subproblems, the Reverse_Distributed_Rolled_Break_Cyclic routine reallocates the data according to a pure cyclic layout.

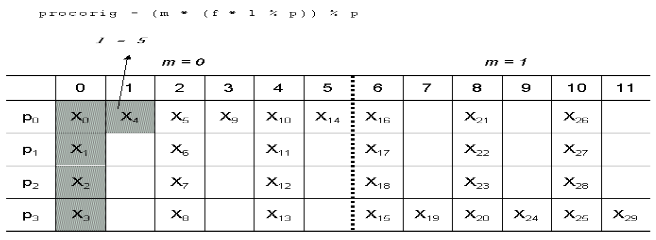
```
Factorize(n, k);     // Compute f[0], ..., f[k-1]
Distribute_Cyclic(); // The permutation of the
                     // input array is distributed
                     // following a cyclic distribution
l = 1;               // l = l[i]; m = m[i]
for (i = 0; i < k; i++) {
  m = n / (f[i] * l);

  Distribute_Rolled_Break_Cyclic(l, m, f);
                      // m[i] * l[i]/p sub-DFTs of size
                      // f[i] have been distributed on
                      // each processor

  for (j = 0; j < m * l /p; j++) {
    v[j] = D * v[j];  // v[j] is the local vector
                      // corresponding to sub-DFT j
                      // D is the diagonal matrix of order
                      // f[i]l. Submatrices of D of order
                      // f[i]f[i] are involved in each
                      // product

    v[j] = Wf * v[j]; // Wf is the DFT matrix of order f
  }
  Reverse_Distributed_Rolled_Break_Cyclic(l, m, f);
  l = l * f[i];
}
```

A total number of lm sub-DFTs of size f must be solved in iteration i. The perfect load balance of the work is found when the lm sub-DFTs can be evenly distributed among the actual number of processors. The `Rolled_Break_Cyclic` distribution groups them into m blocks of size l among the processors. Each block has a representative processor, `procorig`, from which the allocation of data starts in the block. A cyclic assignment is then performed. After l data elements have been allocated, the cyclic distribution breaks and continues again starting from `procorig`. The balanced load of work is attained since `procorig` receives $\lfloor l/p \rfloor + 1$ sub-DFTs and the remaining processors receive $\lfloor l/p \rfloor$ sub-DFTs. Figure 4 depicts a `Rolled_Break_Cyclic` distribution of $n = 30$ elements on $p = 4$ processors. The number of blocks is $m = 2$, and the number of sub-DFTs, per block, $l = 5$, is divided among the processors. In block $m = 0$ processor p_0 is `procorg` and is assigned 2 sub-DFTs, with the first one composed by the elements $\{x_0, x_5, x_{10}\}$ and the second one consisting of $\{x_4, x_5, x_{14}\}$. The remaining processors receive 1 sub-DFT in this block. In block $m = 1$ `procorig` is processor p_3.



	0	1	2	3	4	5	6	7	8	9	10	11
p_0	x_0	x_4	x_5	x_9	x_{10}	x_{14}	x_{16}		x_{21}		x_{26}	
p_1	x_1		x_6		x_{11}		x_{17}		x_{22}		x_{27}	
p_2	x_2		x_7		x_{12}		x_{18}		x_{23}		x_{28}	
p_3	x_3		x_8		x_{13}		x_{15}	x_{19}	x_{20}	x_{24}	x_{25}	x_{29}

Fig. 4. A Rolled Break Distribution of $n = 30$ elements on $p = 4$ processors. $l = 5$, $f = 3$ and $m = 2$

4.3 Experimental Results

The experiments presented in this subsection were performed on a PC Cluster consisting of a 32 Intel Pentium Xeon running at 2.4 GHz, with 1 GByte of RAM memory each, and connected through a Myrinet switch. We have used the MPI implementation GMMPI 1.6.3. Vectors with complex entries randomly distributed were generated as input signals for the algorithm. Two instance problem sizes are considered in the experiments, $n =$

$510,510$ and $n = 1,048,576$. The first one is not a power of two. An interesting feature is shown in Fig. 5, where the variation of the communication cost for the FFT algorithm is reported. For both problem sizes, the communication volume remains almost constant when the number of processors is increased, that is, the total time invested in communications is not incremented as more processors are added. Table 2 shows the execution times for the FFT algorithm. Although the first instance size is not a power of two, the curves show a similar behaviour for both problem dimensions: execution times decrease as more processors are added to the experiment. The load balance achieved by the `Rolled_Break_Cyclic` distribution is strongly dependent on the factors f_i selected so that better performances can be expected if the factorizations are adapted to the problem size and the number of processors but the computational effort may be prohibitive.

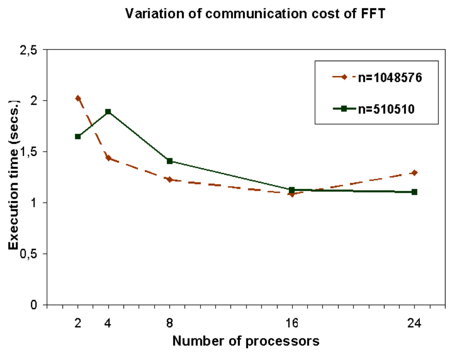


Fig. 5. Variation of the communication cost of the FFT algorithm

Table 2. Time (in sec.) of the parallel implementations for the FFT operation

	Number of processors					
Problem Size	1	2	4	8	16	32
510,510	9.80	8.45	5.27	3.40	2.66	2.16
1,048,576	16.30	14.21	8.95	5.99	4.56	3.67

5 Concluding Remarks

We have described our efforts towards an efficient and portable parallelization of the GSL library using OpenMP and MPI. Experimental results on an SMP and a multicomputer report the performance of several parallel codes for the computation of two classical, simple operations arising in linear algebra and discrete mathematics: the sparse matrix-vector product and the FFT.

In particular, we have experienced that parallelizing the codes for the computation of the sparse matrix-vector product is simple, but the performance depends not only on the algorithm but also on the matrix storage scheme, sparsity pattern, and density rate. Although the scalability of shared-memory architectures is intrinsically limited, experience taught us that in a parallelization of the sparse matrix-vector product on a multicomputer, the communication time plays an important role, becoming a major bottleneck.

We have also observed that the parallelization of the FFT general algorithm is quite elaborate. Although many parallel algorithms have been implemented on dozens of parallel platforms, most of them use internal code optimizations to run efficiently both sequentially and in parallel. Our own FFT codes should then make use of similar optimization techniques to be competitive at the cost of loosing portability. Better performances are expected in shared-memory architectures, where the communication overhead can be avoided, but the load imbalance inherent to the distribution of the sub-DFTs in the parallel algorithm seems difficult to avoid.

References

1. J. Aliaga, F. Almeida, J. M. Badía, V. Blanco, M. Castillo, U. Dorta, R. Mayo, E.S. Quintana, G. Quintana, C. Rodríguez, and F. de Sande. Parallelization of gsl: Architecture, interfaces, and programming models. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *EuroPVM/MPI 2004*, number 3241 in Lecture Notes in Computer Science, pages 199–206. Springer-Verlag, Berlin, Heidelberg, New York, 2004.
2. G. Banga and G. M. Shroff. Communication efficient parallel mixed-radix FFTs. Technical Report 94-1, Indian Institute of Technology, February 1994.
3. I.S. Duff, M.A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms. *ACM Trans. Math. Software*, 28(2):239–267, 2002.
4. M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU scientific library reference manual*, July 2002. Ed. 1.2, for GSL Version 1.2.
5. M. J. Quinn. *Parallel Computing. Theory and Practice*. McGraw-Hill, 1994.
6. C. Temperton. Self-sorting mixed-radix fast fourier transforms. *Journal of Computational Physics*, 52:1–23, 1983.