Parallelization of Hierarchical Matrix Algorithms for Electromagnetic Scattering Problems

Larsson, E.; Zafari, A.; Righero, M.; Francavilla, M.A.; Giordanengo, G.; Vipiana, F.; Vecchi, G.; Kessler, C.; Ancourt, C.; Grelck, C.

Parallelization of Hierarchical Matrix Algorithms for Electromagnetic Scattering Problems

Elisabeth Larsson[1]([✉])[iD], Afshin Zafari[1], Marco Righero[2][iD],
M. Alessandro Francavilla[3], Giorgio Giordanengo[2], Francesca Vipiana[4][iD],
Giuseppe Vecchi[4], Christoph Kessler[5][iD], Corinne Ancourt[6][iD],
and Clemens Grelck[7][iD]

[1] Scientific Computing, Department of Information Technology,
Uppsala University, Uppsala, Sweden
`elisabeth.larsson@it.uu.se, afshin.zafari@gmail.com`
[2] Antenna and EMC Lab (LACE), LINKS Foundation, Turin, Italy
`{marco.righero,giorgio.giordanengo}@linksfoundation.com`
[3] ASML Netherlands BV, Veldhoven, Netherlands
`alessandro.francavilla@asml.com`
[4] Department of Electronics and Telecommunications,
Politecnico di Torino, Turin, Italy
`{francesca.vipiana,giuseppe.vecchi}@polito.it`
[5] Department of Computer and Information Science, Linköping University,
Linköping, Sweden
`christoph.kessler@liu.se`
[6] MINES ParisTech, PSL University, CRI, Paris, France
`corinne.ancourt@mines-paristech.fr`
[7] Informatics Institute, University of Amsterdam, Amsterdam, Netherlands
`c.grelck@uva.nl`

Abstract. Numerical solution methods for electromagnetic scattering problems lead to large systems of equations with millions or even billions of unknown variables. The coefficient matrices are dense, leading to large computational costs and storage requirements if direct methods are used. A commonly used technique is to instead form a hierarchical representation for the parts of the matrix that corresponds to far-field interactions. The overall computational cost and storage requirements can then be reduced to $\mathcal{O}(N \log N)$. This still corresponds to a large-scale simulation that requires parallel implementation. The hierarchical algorithms are rather complex, both regarding data dependencies and communication patterns, making parallelization non-trivial. In this chapter, we describe two classes of algorithms in some detail, we provide a survey of existing solutions, we show results for a proof-of-concept implementation, and we provide various perspectives on different aspects of the problem.

Keywords: Electromagnetic scattering · Hierarchical matrix ·
Task parallel · Fast multipole method ·
Nested equivalent source approximation

1 Introduction

In this chapter, we consider how efficient solution algorithms for electromagnetic scattering problems can be implemented for current multicore-based and heterogeneous cluster architectures. Simulation of electromagnetic fields [42] is an important industrial problem with several application areas. One of the most well known is antenna design for aircraft, but electromagnetic behavior is important, e.g., also for other types of vehicles, for satellites, and for medical equipment. A common way to reduce the cost of the numerical simulation is to assume time-harmonic solutions, and to reformulate the Maxwell equations describing the electromagnetic waves in terms of surface currents [49]. That is, the resulting numerical problem is time-independent and is solved on the surface of the body being studied, see Fig. 1 for an example of a realistic aircraft surface model.

Fig. 1. Surface currents on an aircraft model from a boundary element simulation with around 2 million unknowns.

The size N of the discretized problem, which for a boundary element discretization takes the form of a system of equations with a dense coefficient matrix, can still be very large, on the order of millions of unknowns going up to billions, and this size increases with the wave frequency. If an iterative solution method is applied to the full (dense) matrix, the cost for each matrix-vector multiplication is $\mathcal{O}(N^2)$, and direct storage of the matrix also requires memory resources of $\mathcal{O}(N^2)$. Different (approximate) factorizations of the matrices, that can reduce the costs to $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$, have been proposed in the literature such as the MultiLevel Fast Multipole Algorithm (MLFMA), see, e.g., [43,51]; FFT-based factorization, see, e.g., [50,57]; factorizations based on the Adaptive Cross Approximation (ACA), see, e.g., [67]; or based on H2 matrices as the Nested Equivalent Source Approximation (NESA) [37–39].

All these approximations can be seen as decomposing the original dense matrix into a sparse matrix accounting for near field interactions, and a hierarchical matrix structure with low storage requirements accounting for far field interactions.

A large investment in terms of research and development has been made in constructing and implementing these rather complex algorithms efficiently. A large part of this effort was made before the advent of multicore architectures. Then, the focus was much more on minimizing the amount of computations, optimizing for a powerful server, potentially with a few processors (cores). Now, clusters are built from regular computers with large numbers of cores, and sometimes the additional complexity of accelerators. Memory bandwidth is often the limiting factor in this case. The question for companies as well as researchers with advanced application codes is how to make these codes run on their own or their customers' clusters, with the least effort in terms of changes to the code, and with the maximum output in terms of utilization of the cluster and computational efficiency.

Some of the properties of the hierarchical matrix algorithms that make parallel implementation challenging are first the general form of the algorithm, with interleaved stages of interactions in the vertical direction between parents and children in a tree structure, and horizontally between the different branches at each level of the tree. The stages typically have different levels of parallelism and work loads, and there is a bottleneck when the algorithm reaches the coarsest level of the tree structure, and the amount of parallelism is the smallest. That is, the algorithm itself is generally heterogeneous. Furthermore, the tree can be unbalanced in different ways due to the geometry of the underlying structure, and the groups at the finest level can contain different number of unknowns depending on how the scattering surface cuts through space. An overview of the challenges inherent in the implementation of the fast multipole method (FMM), which is one of the algorithms in this class, on modern computer architectures can be found in [13].

In the following sections, we first provide a high-level description of two types of hierarchical algorithms for electromagnetic scattering. Then in Sect. 3 we provide a survey of literature and software in this area. Section 4 discusses two task-parallel implementations of a simplified algorithm for shared memory. Then in Sect. 5, different perspectives regarding the question of how to eventually port the software to clusters and heterogeneous architecture are given.

2   Two Classes of Algorithms and Their Properties

In this section, we will go deeper into the MLFMA and NESA algorithm classes, and describe their properties from the parallelization perspective. For the mathematical details of the algorithms, see [43] for MLFMA and [37–39] for NESA.

2.1   Interaction Matrices

Solving a system of equations for the unknown surface currents given measured field values can be seen as an inverse problem. When using an iterative method, we transform the inverse problem to repeated solutions of the forward problem, which is easier to address.

The forward problem consists of computing an electromagnetic field given a distribution of sources/charges. We will use two examples to make it more concrete. A well known model problem is to compute the electrostatic potential,

$$\phi(\boldsymbol{x}) = \sum_{j=1}^{N} K(\boldsymbol{x}, \boldsymbol{x}_j) q_j, \tag{1}$$

generated by the point charges q_j located at the points $\boldsymbol{x}_j$. The kernel $K(\cdot, \cdot)$, which is logarithmic in two dimensions and proportional to the inverse distance in three dimensions, represents the interaction between the field points and the charges.

The corresponding scattering problem that is of real interest in industry and for research has a similar structure and consists of computing the electric and/or magnetic fields generated by surface currents on for example a metallic object such as an aircraft. We write this in simplified form as

$$\boldsymbol{E}(\boldsymbol{r}) = \int_{\partial\Omega} \left(G(\boldsymbol{r}, \boldsymbol{r}') \boldsymbol{j}(\boldsymbol{r}') + \frac{1}{k^2} \nabla \left(G(\boldsymbol{r}, \boldsymbol{r}') \nabla \cdot \boldsymbol{j}(\boldsymbol{r}') \right) \right) d\boldsymbol{r}', \tag{2}$$

where $\boldsymbol{r}$ is a point in space, $\partial\Omega$ is the surface of the object, and $G(\cdot, \cdot)$ is a Green's function.

To render the problems tractable for computer simulation, they are discretized. In the first case, we already have discrete charges. In the second case, a boundary integral formulation of the problem is used, where we represent the fields and surface currents by a set of basis functions v_j and corresponding coefficients, which we denote by E_j and q_j, $j = 1, \ldots, N$. Henceforth, we will refer to q_j as sources, and to individual basis functions as locations. The fields are evaluated in the same discrete locations as where the sources are located. This allows us to express the forward problem as a matrix–vector multiplication

$$E = Zq, \tag{3}$$

where E is the vector of the field variables, Z is an $N \times N$ interaction matrix where element z_{ij} describes the contribution from a unit source at location j to the field at location i, and q is the vector of source values.

2.2   The Hierarchical Algorithm

The basis for the fast algorithms is that interactions between locations near to each other are stronger then distant interactions. In the algorithms, near-field interactions are computed directly, while far-field interactions are approximated in such a way that the required storage and the amount of computations is decreased while still respecting a given error tolerance.

The computational domain is hierarchically divided into groups (boxes), which can be represented as an oct-tree (quad-tree in two dimensions) with levels $\ell = L_0, \ldots, \ell_{\max}$. Since the charges are located only on the surface of the

body, many of the groups, especially on the finer levels, are empty of charges, and are pruned from the tree. When we construct the hierarchical algorithm, we consider interactions between groups. We let E_i denote the field variables in group i at level $\ell_{\max}$, and we let E_i^j be the contribution from group j at level $\ell_{\max}$ to E_i, such that

$$E_i = \sum_j E_i^j. \tag{4}$$

Using the direct matrix–vector multiplication, we have that

$$E_i^j = Z_{i,j} Q_j, \tag{5}$$

where $Z_{i,j}$ is a matrix block, and Q_j is the vector of charges in group j at level $\ell_{\max}$. In the hierarchical fast matrix–vector multiplication algorithm, only the near-field interactions are computed directly. These are here defined as interactions between groups that are neighbours at the finest level. The far-field interactions are instead approximated. At each level of the tree structure, starting from level L_0, the far-field groups are identified as those that are not neighbours to the target group. As much of the far-field interaction as possible is treated at each level, to minimize the total number of groups to interact with. In Fig. 2, we show the layout of the near and far-field for computing the field at one location (the black box) in a two-dimensional geometry.
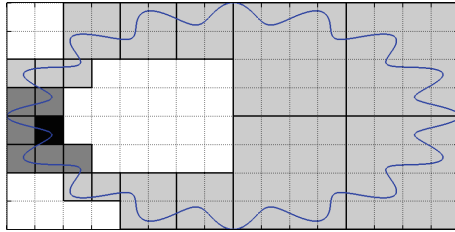
Fig. 2. Illustration of a two-dimensional domain that is hierarchically divided into three levels of boxes. Charges are located on the wavy curve. For the black target box, the near-field consists of the six dark gray neighbouring boxes. The far-field at each level consists of the four large, five medium, and five small light gray boxes that are not neighbours of the target box.

The far-field approximation for one interaction has the following general form

$$E_i^j = Z_{i,j} Q_j \approx R_i \underbrace{\mathcal{P}_{\ell_{\max}-1}^{\ell_{\max}} \cdots \mathcal{P}_{\bar{\ell}}^{\bar{\ell}+1}}_{\text{descending}} \mathcal{T}_{i,j}^{\bar{\ell}} \underbrace{\mathcal{P}_{\bar{\ell}+1}^{\bar{\ell}} \cdots \mathcal{P}_{\ell_{\max}}^{\ell_{\max}-1}}_{\text{ascending}} S_j Q_j, \tag{6}$$

and can be described in terms of the five steps described in Algorithm 1.

Radiation:

The sources Q_j are converted into an intermediate representation $X_j^{\ell_{\max}} = S_j Q_j$.

Source transfer:

The intermediate representation is propagated up through the parent groups $X_j^{\ell-1} = \mathcal{P}_\ell^{\ell-1} X_j^\ell$ until level $\bar{l}$, where the far-field interaction takes place.

Translation:

The far-field interaction between groups i and j is computed at level $\bar{\ell}$. The result is an intermediate field representation $Y_i^{\bar{\ell}} = \mathcal{T}_{i,j}^{\bar{\ell}} X_j^{\bar{\ell}}$.

Field transfer:

The intermediate field representation is propagated down through the child groups $Y_i^{\ell+1} = \mathcal{P}_\ell^{\ell+1} Y_i^\ell$ until the finest level is reached.

Reception:

The intermediate field representation is evaluated at the actual field locations, $E_i^j = R_i Y_i^{\ell_{\max}}$.

Algorithm 1. The algorithm for computing one interaction term.

If we change the view, and instead see the algorithm from the perspective of one particular group at level ℓ that takes part in several interactions, the work related to that group can be expressed as Algorithm 2.

Upward Phase:

if $\ell = \ell_{\max}$ **then**

Compute $X_j^{\ell_{\max}} = S_j Q_j$ for local sources.

else

Receive source contributions from all child groups.

end

if $\ell > L_0$ **then**

Send accumulated source contribution to parent.

end

Translation Phase:

Compute $Y_i^\ell = \mathcal{T}_{i,j}^\ell X_j^\ell$ according to interaction list.

Downward Phase:

if $\ell > L_0$ **then**

Receive field contribution from parent.

end

if $\ell < \ell_{\max}$ **then**

Send field contribution to all child groups.

else

Compute $E_j = R_j Y_j^{\ell_{\max}}$ for local target points.

end

Algorithm 2. The algorithm seen from the view of one particular group.

For the parallel implementation, there are several relevant aspects to keep in mind. For the upward and downward phases, communication is performed vertically in the tree, between parent and child groups. The translation operations on the other hand need horizontal communication. Due to the hierarchical structure, each group has an interaction list of limited size. The three phases of the algorithm can be overlapped, since different groups complete the phases at different times. Even more important is that the near-field interactions for disjunct groups are independent and can be interspersed with the far-field computations.

The memory savings that the fast algorithms provide stem from the fact that the far-field part of the interaction matrix is replaced with the operators in (6). These are the same for groups that have the same position relative to each other. That is, only a limited number of operators are needed at each level.

2.3   Specific Properties of the NESA Algorithm

In the NESA algorithm, all of the far-field operations consist in expressing sources and field in terms of equivalent charges. The actual sources in a group at level $\ell_{\max}$ can through a low rank approximation be represented by a set of equivalent sources that generate a matching field at some control points located at an exterior test surface. In the same way, the equivalent sources in a child group can be represented by another set of equivalent sources at the parent group. This is schematically shown for a two-dimensional problem in Fig. 3. The number of equivalent charges Q is the same in each group, which is why we can save significantly in the far-field computation. The translation and field transfers are managed similarly. We will not go into all details here, instead we refer to [37].
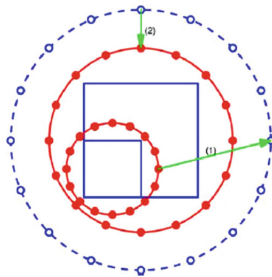
Fig. 3. A parent group and one of its children are illustrated. The points on the solid circles are where the equivalent sources are located, and the points on the dashed circle are where the fields are matched.

To understand the computational properties of the NESA algorithm, we characterize each operation in terms of how much memory it needs to load counted in double precision numbers, and how many floating point operations (flop) are performed. We also provide the computational intensity, in flop/double. The results

Table 1. Characterization of the matrix–vector products in the NESA algorithm. The number of sources in group j is denoted by n_j.

Operator	Data size	Compute size	Intensity
Near field			
$Z_{i,j}$	$n_i \times n_j$	$2n_i n_j$	2
Far field			
S_j	$Q \times n_j$	$2n_j Q$	2
$\mathcal{P}_\ell^{\ell-1}$	$Q \times Q$	$2Q^2$	2
$\mathcal{T}_{i,j}^\ell$	$Q \times Q$	$2Q^2$	2
$\mathcal{P}_\ell^{\ell+1}$	$Q \times Q$	$2Q^2$	2
R_i	$n_i \times Q$	$2n_i Q$	2

are listed in Table 1. All of the operations in the NESA algorithm are dense matrix–vector products, with the same computational intensity of 2 flop/double. For modern multicore architectures, a computational intensity of 30–40 is needed in order to balance bandwidth capacity and floating point performance, see for example the trade-offs for the Tintin and Rackham systems at UPPMAX, Uppsala University, calculated in [64]. This means that we need to exploit data locality (work on data that is cached locally) in order to overcome bandwidth limitations and scale to the full number of available cores.

2.4   Specific Properties of the MLFMA Algorithm

In the MLFMA algorithm, the intermediate representation of sources and fields is given in terms of plane wave directions $\hat{\kappa} = (\theta, \phi)$, where θ is the polar angle, and ϕ is the azimuthal angle in a spherical coordinate system. When computing far-field interactions, the Green's function can be represented using an integral over the directions, which numerically is done through a quadrature method. The accuracy of the approximation depends on the number of directions that are used. A difference compared with the NESA method is that the number of directions that are needed scale with the box size.

Table 2. An example of the number of directions N_ℓ needed at each level in the MLFMA algorithm, starting from the finest level $\ell = \ell_{\max}$.

$\ell_{\max} - \ell$	0	1	2	3	4	5	6	7	8	9
L_ℓ	5	7	10	15	23	38	66	120	224	428
N_ℓ	72	128	242	512	1152	3042	8978	29282	101250	368082

To compute the number of directions needed for a box at level ℓ, we first compute the parameter L_ℓ from the wave number of the electromagnetic wave,
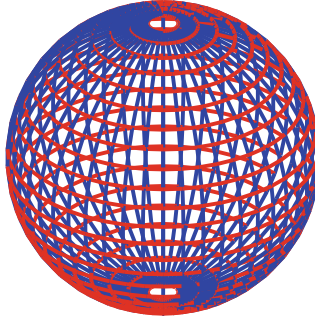
Fig. 4. A unit sphere discretized for $L_\ell = 23$ with 24 points in θ (latitudes) and 48 points in ϕ (longitudes).

the diagonal d_ℓ of the box, and the desired error tolerance τ [43, Sect. 5.3]. Then the box is discretized with $L_\ell + 1$ points in the θ-direction and $2L_\ell + 2$ points in the ϕ-direction giving a total number of $N_\ell = 2L_\ell^2 + 4L_\ell + 2$. Using a realistic tolerance $\tau = 1e - 4$ and an appropriate box size for the finest level leads to the sequence of sizes given in Table 2. Figure 4 shows the discretized sphere for $L_\ell = 23$. The wide range of sizes for the representations at different levels does pose a challenge for parallel implementations.

The interpolation step between parent and child or vice versa can be realized in different ways. Here, we consider the Lagrange interpolation method described in [43]. Then the value at one point at the new level is computed using the m nearest neighbours in each coordinate direction. The operations of one interpolation step are shown schematically in Fig. 5.
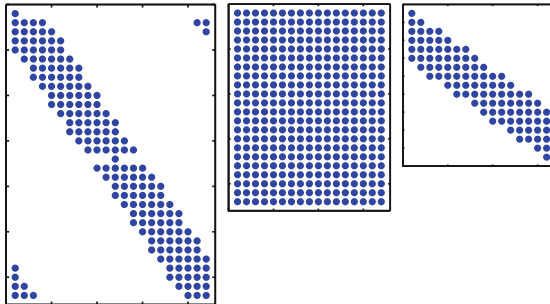
Fig. 5. To interpolate the data (middle) from a child $L = 10$ to a parent $L = 15$, a sparse interpolation matrix (left, right) is applied to each of the data dimensions. The matrix sizes are here 32×22, 22×17, and 17×16. The data matrix is extended with $m/2$ columns to each side to manage the periodicity at the poles.

Similarly as for the NESA algorithm in the previous subsection, we characterize the work performed during the algorithm and evaluate its computational

Table 3. Characterization of the steps in the MLFMA algorithm. The number of sources in group j is denoted by n_j.

Operator	Data size	Compute size	Intensity
Near field			
$Z_{i,j}$	$n_i \times n_j$	$2n_i n_j$	2
Far field			
S_j	$N_\ell \times n_j$	$2n_j N_\ell$	2
$\mathcal{P}_\ell^{\ell-1}$	$(2L_\ell \times L_\ell)$	$2m\left(2L_\ell L_{\ell-1} + 2L_{\ell-1}^2\right)$	$2m\left(\frac{L_{\ell-1}}{L_\ell} + \left(\frac{L_{\ell-1}}{L_\ell}\right)^2\right)$
$\mathcal{T}_{i,j}^\ell$	$2\left(2L_\ell \times L_\ell\right)$	$\left(2L_\ell^2\right)$	0.5
$\mathcal{P}_\ell^{\ell+1}$	$(2L_\ell \times L_\ell)$	$2m\left(2L_\ell L_{\ell+1} + 2L_{\ell+1}^2\right)$	$2m\left(\frac{L_{\ell+1}}{L_\ell} + \left(\frac{L_{\ell+1}}{L_\ell}\right)^2\right)$
R_i	$n_i \times N_\ell$	$2n_i N_\ell$	2

intensity. The results are given in Table 3. The radiation and reception steps are matrix–vector products also in this case. The interpolation steps have a higher computational intensity. For $m = 6$ and the child to parent operation, we get 40–66 flop/data, while for the parent to child operations, we get 10–15 flop/data. The translation step is often a bottleneck in parallel implementation. It is an elementwise multiplication with an intensity less than one flop/data.

3   State of the Art

There is a rich literature on parallel implementation of hierarchical matrix algorithms. Many of the implementations are aimed at volume formulations (particles/charges are located in a volume), as opposed to surface formulations as for the scattering problem. The volume formulation is more likely to have a large number of particles in a group, and a more well-balanced tree structure.

The most common parallelization approach, targeting distributed memory systems, is to partition the tree data structure over the computational nodes, and use an MPI-based parallelization [34]. The resulting performance is typically a bit better for volume formulations then for boundary formulations, since the computational density is higher in the former case. A particular issue for the MLFMA formulation of electromagnetic scattering problems is that the work per element (group) in the tree data structure increases with the level, and additional partitioning strategies are needed for the coarser part of the structure [6,30,56].

The ongoing trend in cluster hardware is an increasing number of cores per computational node. When scaling to large numbers of cores, it is hard to fully exploit the computational resources using a pure MPI implementation, due to the rapid increase in the number of inter-node messages with the number of MPI processes for communication heavy algorithms [64]. As is pointed out in [35], a hybrid parallelization with MPI at the distributed level and threads within the computational nodes is more likely to perform well. That is, a need for

efficient shared memory parallelizations of hierarchical algorithms to be used in combination with the distributed MPI level arises.

The emerging method of choice for implementing complex algorithms on multicore architectures is dependency-aware task-based parallel programming, which is available, e.g, through the StarPU [5], OmpSs [46], and SuperGlue [54] frameworks, but also in OpenMP, since version 4.0. Starting with [7], where StarPU is used for a task parallel implementation of an FMM algorithm, several authors have taken an interest in the problem. In [31], SuperGlue is used for a multicore CPU+GPU implementation of an adaptive FMM. The Quark [61] run-time system is used for developing an FMM solver in [40]. Since tasks were introduced in OpenMP, a recurring question is if the OpenMP implementations can reach the same performance as the specific run-times discussed above. An early OpenMP task FMM implementation is found in [2]. This was before the depend clause was introduced, allowing dependencies between sibling tasks. OpenMP, Cilk and other models are compared for FMM in [66], OpenMP and Klang/StarPU are compared in [1], and different OpenMP implementations and task parallel run-times are compared with a special focus on locking and synchronization in [4]. A common conclusion from these comparisons is that the commutative clause provided by most task parallel run-time systems is quite important for performance, and that this would be a useful upgrade of OpenMP tasks for the future.

An alternative track is to develop special purpose software components for the class of FMM-like algorithms, see, e.g., PetFMM [12] and Tapas [21].

An open source implementation of MLFMA is available through the Puma-EM software [47], parallelized with MPI. An example of a commercial MLFMA software is Efield [16] provided by ESI Group, parallelized for shared memory.

4   Proposed Solution and Proof of Concept

During the last decade task-parallel programming has emerged as the main programming paradigm to run scientific applications on modern multicore- and heterogeneous computer architectures. A recent and fairly complete overview of the current state of the art can be found in [52].

The key idea is that the programmer provides the sequential work-flow of an algorithm in terms of tasks. These are then submitted to a run-time system, which analyses the data dependencies of the tasks and schedules them onto available hardware resources to be executed in parallel. It can in some cases be possible to obtain higher performance by hand-tuning a code, but the cost in programming effort and the renewed cost if the system configuration changes are usually considered too high.

There are several arguments for using task parallel programming for the hierarchical matrix–vector products considered here.

– The work flow is already described in terms of *tasks* operating on data associated with the individual groups. Therefore, the overhead of converting the algorithm into a suitable form can be largely avoided.

– The data size varies between groups, the number of child groups and interactions across the tree structure. The amount of work varies with the level and the phase of the algorithm. All of this indicates that the asynchronous task execution provided by a run-time system is more likely to be efficient than a statically determined schedule.
– The dependencies between tasks are complex, problem dependent, and hard to analyze manually. With the run-time scheduling, dependencies are automatically managed, and tasks can run as soon as their dependencies have been met. Furthermore, the run-time guarantees correctness in the sense that if the sequential task flow of the application code is described correctly, the parallel execution guarantees to respect the order up to admissible interleavings.

As a proof of concept, we have implemented the NESA algorithm for the electrostatic potential problem using the SuperGlue [54] framework. A detailed description of the implementation details and the results can be found in [63]. A benefit of using the NESA algorithm is that the tasks are more similar both in size and type than for the MLFMA algorithm. The main arguments for choosing the SuperGlue framework are (i) that it has very low scheduling overhead, and can therefore handle small task sizes well, and (ii) that commutative data accesses are naturally included in the dependency management based on data-versioning. Commutative accesses relate to tasks that touch the same data, and that can therefore not run concurrently, but that can otherwise run in any order.

We have also implemented the NESA algorithm using OpenMP tasks, and provide some results and comments on how the two implementations compare.

In the following subsections, we provide a brief survey of task parallel programming frameworks, we discuss the SuperGlue and OpenMP implementations, and we provide some illustrative performance results.

4.1 A Task-Parallel Programming Overview

One of the key features of task parallel programming is that it makes it relatively easy for the programmer to produce a parallel application code that performs well. However, it is still important for the programmer to understand how to write a task parallel program and how various aspects of the algorithm are likely to impact performance.

The granularity of the tasks determines the number of tasks, which has a direct effect on the potential parallelism. As an application programmer, it is beneficial to be aware of how tasks interact with each other and with the data. That is, to understand the character of the data dependencies. There may be different ways of splitting the work that lead to different degrees of parallelism. In the NESA and MLFMA cases, a basic task size is given by the algorithm through the division of the domain into groups. The discussion to have is whether some groups need splitting (the coarse levels in MLFMA) or merging (the leaf groups).

The granularity of tasks also has an effect on how the tasks interact with the memory hierarchy. If the tasks are small enough, data may fit into the cache. If the run-time system is locality-aware such that tasks are scheduled at the

cores where the data they need is cached, significant performance gains may be secured. As was discussed in Sects. 2.3 and 2.4, the computational intensity provided by the tasks of the NESA and MLFMA algorithms is not enough to scale well if all of the data is read from the main memory.

In [55] resource-aware task-scheduling is investigated. It is shown that the effect of, e.g, bandwidth contention between tasks can be reduced by co-scheduling a mix of diverse tasks. However, in the NESA case, all of the tasks have a similar computational intensity, so that approach is not applicable.

From the user perspective, it would be ideal if there were only one framework for task parallelism, or at least one common standard for task parallel programming implemented by different frameworks. Steps are being taken in this direction, see also the implementation in [62], but it will take some time until it is in place. Meanwhile, we provide an overview of some of the more relevant initiatives.

The StarPU framework [5,52] was initially developed to manage scheduling between the CPU and GPU resources in one computational node. It has over time been developed in different ways and has become one of the most widely adopted general purpose run-time systems. In StarPU, an important component is the management of data transfers and data prefetching. Advanced performance prediction based on performance measurements is used in the different scheduling algorithms. StarPU has very good performance for large scale problems with relatively large task sizes. When task sizes become too small, the overhead of the advanced scheduling is too large, and performance goes down.

Another important run-time system is OmpSs [15], which is the current representative of the StarSs family [46]. In OmpSs, the tasks are defined through compiler directives in the same way as in OpenMP. In fact, the development of the OpenMP standard in terms of tasks and task dependencies is driven by the development of OmpSs. In this way, the constructs and implementations are well tested before being adopted by the standard. The use of directives can be seen as less intrusive when transforming legacy code into task parallel code compared with the use of specific APIs for task submission.

LAPACK [3], which implements a large selection of linear algebra operations, is one of the most widely used libraries in scientific computing. With the advent of multicore architectures, a number of projects were started to provide multicore and GPU support. These have now converged into using the PaRSEC run-time system [8], which has excellent performance both for large and small task sizes. PaRSEC can be used for all types of algorithms, but it requires the task dependencies to be expressed in a specific data flow language. This allows to build a parametrized task graph that can be used efficiently by the run-time system, but it can be an obstacle for the application programmer.

The SuperGlue framework [54] was developed mainly for research purposes with a focus on performance. It is a general-purpose task parallel framework for multicore architectures. It is very lightweight, it uses an efficient representation of dependencies through data versions, and has very low overhead, such that comparatively small tasks can be used without loosing performance.

Tasks with dependencies were introduced in OpenMP 4.0. The dependencies are only between sibling tasks submitted in the same parallel region, and there is not yet support for commutative tasks, which are relevant for the NESA and MLFMA types of algorithms. The main reason for using OpenMP is that it is a standard and is likely to remain, making it a relatively secure investment in coding.

4.2   The SuperGlue Task Parallel Implementation

SuperGlue is implemented in C++ as a headers only library. In order to write a task-based version of the NESA algorithm for SuperGlue, we need to define a SuperGlue task class for the matrix–vector product that is the computational kernel used in all the tasks. In Program 1.1 we show a slightly simplified code that emphasizes the most relevant parts. The task class contains the data that is touched by the task, a constructor, and a run method. In the constructor, the types of data accesses are registered. In this case, it is a commutative (add) access to the output vector. The read accesses to the input data are not registered as that data is not modified during execution. The access information is used for tracking dependencies, and extra dependencies increase the overhead cost.

The constructor is called at task submission, while the run method is called at task execution time.

```
1  class SGTaskGemv : public Task<Options,3>{
2  private:
3    SGMatrix *A,*x,*y;
4  public:
5    SGTaskGemv(SGMatrix &A_, SGMatrix &x_, SGMatrix &
        y_)
6    {
7      A = &A_;
8      x = &x_;
9      y = &y_;
10     Handle<Options> &hy = y->get_handle();
11     register_access(ReadWriteAdd::add, hy);
12   }
13
14   void run(){
15     double *Mat= A->get_matrix()->get_data_memory();
16     double *X   = x->get_matrix()->get_data_memory();
17     double *Y   = y->get_matrix()->get_data_memory();
18     cblas_dgemv(Mat, X, Y);
19   }
20 };
```

Program 1.1. The MVP task class

In the application code all former calls to the matrix–vector product subroutine should be replaced by the corresponding task submission. If we hide the task submission statement in a subroutine, the syntax of the application code does not need to change at all. The new subroutine that replaces the original matrix–vector product by the task submission is provided as Program 1.2.

```
1 void gemv(SGMatrix &A, SGMatrix &x, SGMatrix &y){
2     SGTaskGemv *t= new SGTaskGemv(A, x, y);
3     sgEngine->submit(t);
4 }
```

Program 1.2. The subroutine that submits an MVP task.

There are also other small changes such as starting up the SuperGlue runtime, and the SGMatrix data type, which equips the 'ordinary' matrix type with the data handle that is used when registering accesses. A longer description of the implementation can be found in [63], and the full implementation is available at GitHub[1].

4.3   The OpenMP Task-Parallel Implementation

An implementation with a similar functionality as the task-parallel implementation described above can—with some care—be created with OpenMP as well. A simple *task* construct was introduced in OpenMP 3.0, and a *depend* clause was added in OpenMP 4.0, to allow dependencies between sibling tasks, i.e, tasks created within the same parallel region. This means that if we create several parallel regions for different parts of the algorithm, there will effectively be barriers in between, and the tasks from different regions cannot mix.

```
1 #pragma omp parallel
2 {
3   #pragma omp single
4   {
5     // Submit tasks for near-field multiplication
6     FMM::mv_near_field(OT, C, Q);
7     // Submit tasks for far-field multiplication
8     FMM::mv_far_field(OT, C, Q);
9   }
10 }
11 #pragma omp taskwait
12 #pragma omp barrier
```

Program 1.3. The structure of the OpenMP implementation. There is one global parallel region (lines 1–10), and within this region only one thread can submit tasks (lines 3–9).

---

[1] https://github.com/afshin-zafari/FMM/.

The proper way to do it is to create one parallel region that covers the whole computation, and then make sure that only one thread generates tasks such that the sequential order is not compromised. An excerpt from the OpenMP main program that illustrates this is shown in Program 1.3. The tasks are implicitly submitted from the near-field and far-field subroutines, whenever the `cblas_dgemv` subroutine is invoked.

The tasks are defined using the task pragma with the depend clause, see Program 1.4. Only the (necessary) inout dependence for the output data vector is included. Adding the (nonessential) read dependencies on the matrix and input data vector was shown in the experiments to degrade performance.

```
1 #pragma omp task depend(inout:Y[0:N])
2 cblas_dgemv(Mat, X, Y);
```

Program 1.4. The OpenMP task pragma that defines a gemv task.

As can be seen, the implementation is not so difficult, but there are several ways to make mistakes that lead to suboptimal performance. The programmer needs to understand how the task generation, the task scheduling, and the parallel regions interact.

4.4   Performance Results

In this section we summarize the experimental results from [63] and relate these to the arguments we gave for using a task-based parallel implementation. The ease of implementation was discussed in the previous two subsections. The next two arguments concerned the benefits of asynchronous task execution, dynamic and automatic scheduling, and mixing of computational phases.

Execution traces for the SuperGlue implementation, when running on one shared memory node of the Tintin cluster at the Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX), are shown in Fig. 6. The simulation parameters P and Q are the average number of sources in one group at the finest level (the average of n_j), and the number of auxiliary sources in each group, respectively. The near-field trace (top) nicely illustrates how tasks of different sizes are scheduled asynchronously onto 16 worker threads with no visible idle time between the tasks. The far-field trace furthermore illustrates that the different computational phases can be interleaved to a large extent using a schedule that it would be difficult to construct statically. Finally the last trace shows that the far-field tasks can be embedded in the near-field computation. As will be discussed below, this is beneficial since the far-field tasks have a lower computational intensity, and in this case are also smaller. The idle time that can be seen in the beginning for thread 0 in the middle and bottom panels is the time for task submission.

Another question that was investigated using the proof of concept implementation was how the task size impacts scalability, and how small tasks can be used without loosing performance. The same problem with $N = 100\,000$ source points is solved in all experiments, but the method parameters P (the average number
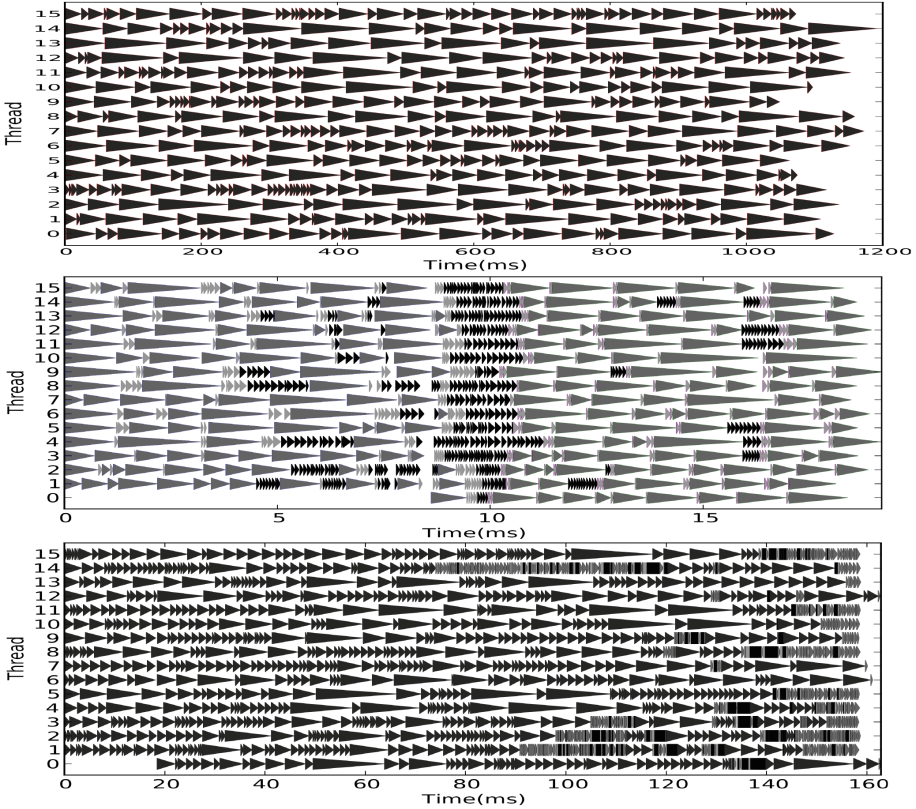
Fig. 6. Execution traces for the near field computation (top), the far field computation (middle), and the combined execution (bottom) for $Q = 100$ and $P = 400$. Each task is shown as a triangle, and the color indicates which phase of the algorithm it belongs to. Near-field (dark gray), radiation (medium gray), source transfer (light gray), translation (black), field transfer (light gray), and reception (medium gray).

of source points at the finest level) and Q (the number of auxiliary points used for each group) are varied between the experiments.

We compute the speedup S_p using p cores as $S_p = T_1/T_p$. Each node of the Tintin cluster consists of two AMD Opteron 6220 (Bulldozer) processors. A peculiarity of the Bulldozer architecture is that each floating point unit (FPU) is shared between two cores. This means that the theoretical speedup when using $2p$ threads (cores) is only p, and the highest theoretical speedup on one node with 16 threads is 8.

Figure 7 shows the results for different task sizes. The near-field computation scales relatively well for all of the task sizes, but the performance improves with size P. For the far-field, there is no scalability when both P and Q are small. The situation improves when the sizes increase, but the scalability is significantly worse than for the near-field. For the combined computation, the results are

better than the far-field results with the same sizes, and for the larger tasks even better than the near-field results. That is, the mixing of the two phases allows the limited scalability of the far-field computation to be hidden behind the better performance of the near-field computations. We can however conclude that $Q = 10$ and $P = 50$, which are reasonable numbers for the two-dimensional case results in tasks that are too small for scalability. Using $Q = 100$, which is suitable for the three-dimensional problem, is however enough for shared memory scalability. This is an indication that the proof of concept approach can be used for the real three-dimensional problem.
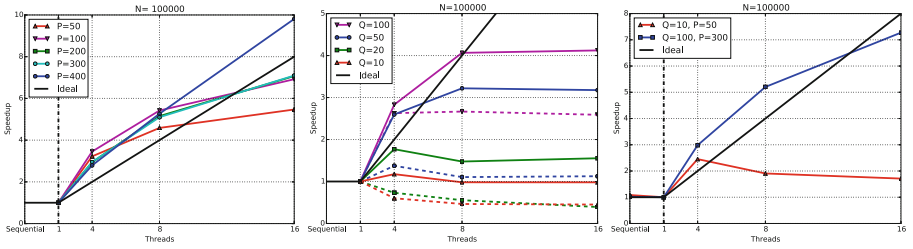
Fig. 7. Speedup for the near-field computation (left), the far-field computation (middle) for $P = 400$ (solid lines) and for $P = 50$ (dashed lines), and the combined computation (right).

In Table 4, we compare the execution times, the speedup, and the utilization for execution with small tasks and with larger tasks. The utilization is defined as the fraction of the total execution time that is spent in executing tasks. That

Table 4. The parallel execution time T_p, the speedup S_p, the speedup in relation to the theoretical speedup S_p^*, and the utilization U_p computed as the fraction of time spent executing tasks, for two problem settings.

p	T_p [ms]	S_p	S_p/S_p^*	U_p
$Q = 10$, $P = 50$				
1	244	1.0	1.00	0.90
4	111	2.2	1.10	0.55
8	137	1.8	0.44	0.29
16	156	1.6	0.20	0.21
$Q = 100$, $P = 300$				
1	1192	1	1.00	0.99
4	401	3.0	1.49	0.98
8	228	5.2	1.31	0.98
16	163	7.3	0.92	0.96

is, the lack of utilization reveals overhead, idle time and load imbalance. For the problem with larger tasks, both utilization and speed are close to optimal. For the problem with small tasks, the utilization goes down to 21% for 16 threads. Then one might expect that the execution time $T_{16} = T_1/16/0.21$, leading to a speedup $S_{16} = 3.4$, but this is not at all the case. Figure 8 shows the slowdown of individual task execution as a function of the number of threads. A factor of 2 is expected for 16 threads due to the Bulldozer architecture. This is also the case for the larger tasks. For the smaller tasks, the far-field computations exhibit a slowdown of 4, which limits the potential scalability to maximum 4 at 16 threads. The computational intensity does not change with the task size, but a potential explanation can be found when looking at the scheduling in the run-time system. For large enough tasks, the run-time system has time to use the knowledge of which data is needed by a task to place it in the work queue of the thread where that data is cached, thereby ensuring data locality. However, for too small task sizes, task execution becomes faster than task submission, and the opportunity to find the next task 'in time' is lost. Then the threads try to steal work from each other. This results in contention on the work queues as well as a loss of data locality.
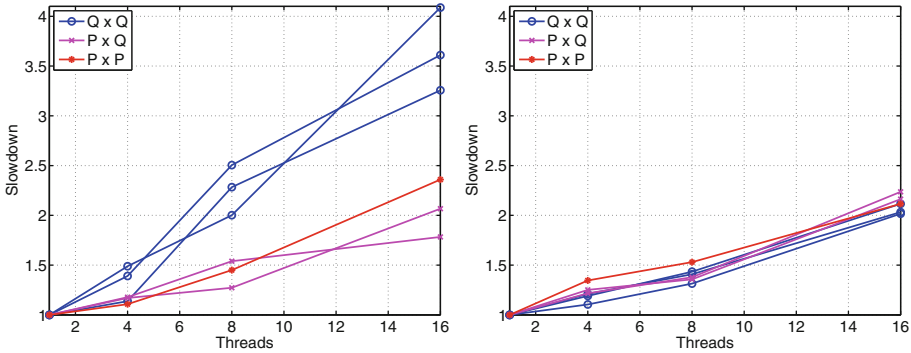
Fig. 8. Increase in individual task execution times for the complete execution for $P = 50$, $Q = 10$ (left) and for $P = 300$, $Q = 100$ (right).

The final question we ask in this section is whether OpenMP is efficient enough to use for this problem. We already mentioned the fact that OpenMP currently does not support commutative tasks. The performance of the OpenMP run-time implementations has increased over time, and will most likely continue to do so.

The experiments were carried out both on a node of the Tintin cluster, described in the previous section, and on a local shared memory system with 4 sockets of Intel Xeon E5-4650 Sandy Bridge processors, yielding a total of 64 cores. On Tintin, the codes were compiled with gcc version 4.9.1 and OpenMP 4.0, while on Sandy Bridge the compiler was gcc 6.3.0 combined with OpenMP 4.5.

We compare the execution times using SuperGlue (SG) and using OpenMP (OMP) for the full execution and for two different task sizes. The results, given in Tables 5 and 6, show that OpenMP is slower for small task sizes, and especially when small sizes are combined with large numbers of threads. However, for the larger problem sizes, the differences are small 5–10%, and the results vary between the two hardware systems. We do not see the effect of the missing commutative clause. As long as tasks are large enough. These results indicate that OpenMP can be used for this type of problem.

Table 5. Execution times in ms for the SuperGlue (SG) and OpenMP (OMP) implementations executed on a Tintin node.

p	$P = 50, Q = 10$			$P = 300, Q = 100$		
	SG	OMP	OMP/SG	SG	OMP	OMP/SG
1	244	285	1.17	1192	1186	1.00
4	111	134	1.21	363	345	0.95
8	137	110	0.80	210	186	0.89
16	156	254	1.63	145	139	0.96

Table 6. Execution times in ms for the SuperGlue (SG) and OpenMP (OMP) implementations executed on the Sandy Bridge system.

p	$P = 50, Q = 10$			$P = 300, Q = 100$		
	SG	OMP	OMP/SG	SG	OMP	OMP/SG
1	438	476	1.09	2318	2556	1.10
4	166	260	1.57	811	913	1.13
8	100	197	1.97	422	469	1.11
16	107	170	1.59	244	253	1.04
32	135	237	1.76	154	157	1.02
64	141	535	3.79	127	133	1.05

5   Perspectives and Future Directions

Our proof-of-concept implementation demonstrates that a task parallel implementation provides the expected benefits. As long as the task granularity is not too small relative to the overhead of the run-time system the proposed solution performs well. Thus, we can recommend this general direction of parallelization, but there are many further aspects to consider; we discuss some of them in the following subsections.

5.1   Recommendations for a Task Parallel 3-D Implementation

When performing large scale three-dimensional simulations, it becomes necessary to use distributed computer systems, and hence distributed parallel programming (or a partitioned global address space (PGAS) model). In [64] it was shown that a hierarchical task-parallel programming model was beneficial for the distributed implementation. Larger tasks are communicated between computational nodes, and then split into subtasks that are executed in parallel within each node.

For the upward and downward phases it seems natural to let a larger task represent operations within a subtree. For the communication-intensive translation phase, it is less clear what the best type of task is. Perhaps translations between subtrees can be performed as one larger task, but this reduces the opportunities to interleave the translation stage with the other stages.

The partitioning of the global tree structure into subtrees would be performed at a level where the number of groups is at least equal to the number of computational nodes. Then the question is how to share the work and the data for the levels above the splitting point. For the NESA algorithm, this is not such a big problem as the amount of work at these lower levels is small. However, for the MLFMA algorithm, the work increases significantly for the lower levels, as can be seen in Table 2. In this case, the work for these levels needs to be divided between the computational nodes, while the data could potentially be shared by all. A drawback of such an approach could be that this kind of splitting becomes more intrusive from the programming perspective, than just making each subroutine call into one task.

As we saw in the proof-of-concept implementation, small task sizes can also become a problem, but from the programming perspective we do not want to explicitly merge work into larger tasks. In a preliminary implementation, which is not yet finished, we performed experiments with batching of small tasks. When tasks are submitted to the run-time system, they are saved in a buffer until there is enough work to actually start a batched task, which then executes all of them at once.

The question of which run-time system or programming model to use is a difficult one. Especially for a company, it is important to know what kind of long-term support of a programming model can be expected, and whether permissions and licenses for using it remain stable. This would be an argument for using OpenMP for the shared memory part. For distributed task-parallel programming, however, there is no similarly established standard as of yet. The choice is then to either develop a custom run-time which is unlikely to be as good as the already existing ones, or to trust an existing one, which may at some point no longer be supported.

5.2   Automatically Mapping Workloads to Accelerators

Applications that perform regular computations on a large number of data are often good candidates for efficient execution on accelerators such as GPUs. However, mapping some parts of the applications onto a GPU is not easy, especially

when the application is written in C++. Indeed, in C++ references to array elements or certain values such as loop bounds can be hidden in function calls. Automatic tools that detect data dependencies statically and generate parallel code and GPU kernels need this information explicitly. Otherwise, dynamic analysis and code instrumentation are required.

Initially, the mapping consists in detecting loops that meet the criteria of the accelerator. These criteria express the adequacy between the loop nest patterns and the target accelerator hierarchy: external parallel loops will be mapped directly to streaming cores and sequential internal loops in threads. The loop nest sizes must be large enough to compensate for communication time and less than the number of possible accelerator threads. Finally, an estimation of the kernel memory footprint is required to fit the overall memory of the GPU.

If we take into account only the pieces of application that naturally respect these constraints, we miss many pieces of code that can benefit from optimization. Gouin presents a methodology to increase the number of application pieces that can benefit from accelerator optimization and describes all necessary mapping stages [24, 25].

The actual programming of GPU kernels, preferably specified within the same source file as the calling CPU code, and of the necessary device memory management and data transfers to/from GPU device memory can be made easier for the programmer by adopting a high-level parallel programming model supporting GPU execution. For example, *OpenACC* allows to write kernels by annotating sequential loop-based code in a style similar to OpenMP parallel loop annotations. The OpenMP task model supports code generation for GPU execution of tasks since *OpenMP 4.0/4.5* with the introduction of the `target` directive for offloading computations to accelerator devices. *SYCL* (https://www.khronos.org/sycl) is a high-level programming layer atop OpenCL that provides a single-source abstraction for OpenCL based accelerator programming. For improved programmability, task-based runtime systems for heterogeneous programming such as StarPU can also be coupled with higher-level programming abstraction layers such as *SkePU* [14], which, from high-level constructs such as skeleton function calls, automatically generate the API calls for the management of tasks, data buffers and their dependencies by the runtime system.

5.3   Optimizing the Task Sizes

In the application the basic task size is given by the algorithm through the division of the domain into groups. As the tiling transformation makes it possible to optimize task granularity at the loop level, adjusting task and group sizes can:

– improve data locality
– improve cache reuse and
– reduce communication overhead.

The new decomposition must be performed in order to balance computations and communications. Considering the OpenMP implementation and a large

number of small tasks, merging could also reduce the global thread creation over-heads and thread scheduling run-time.

Task and group sizes are multi-dimensional spaces and the optimal decomposition parameters depend on the target architecture constraints (memory size, number of cores). Finding these optimal parameters is complex since they are dynamic variables. Autotuner techniques combining profiling information might be used to develop heuristics and to limit the maximum task sizes at each level of the application.

5.4   Limiting Recursive Task Creation on CPU

Task-based computations over recursively defined sparse hierarchical domains such as quadtrees/octrees could, if applicable for the underlying computational problem, choose to stop the recursive subdivision at a certain depth limit and, for example, switch to computations over dense representations below this limit or sequentialize the independent subcomputations instead of creating a smaller task for each of them. For example, OpenMP 4.x provides the `if` clause to the `task` construct for conditional task creation. Such cut-off depth/condition, as well as the degree of task unrolling in general, can be used as a tuning parameter to balance the trade-off between computational work to perform, degree of data parallelism in tasks, and tasking and synchronization overhead. For example, Thoman *et al.* [53] describe a combined compiler and runtime approach for adaptive granularity control in recursive CPU task-parallel programs.

5.5   Techniques for Task and Data Granularity Adaptation on GPU

The task granularity in dynamically scheduled task-based computations on a heterogeneous system can have a major impact on overall performance. Each task executing on an accelerator typically contains just one kernel call, or possibly several kernel calls that execute in sequence on the same accelerator unit. For the application considered in this chapter, tasks/kernels of size 50×50 turn out to be too fine-grained for GPU execution in practice, as most of the GPU's computation capacity remains unused and the task management overhead (which is for StarPU in the order of several dozen microseconds) becomes large in relation to the task's work.

A number of task-based programming environments allow to control task granularity already at task creation, in particular for CPU-based tasks by introducing conditions for recursive task creation, as described in Sect. 5.4.

Moreover, a number of static and dynamic techniques exist for adapting task granularity in a GPU execution context. In the remainder of this section we review a number of such granularity adaptation techniques specifically for GPU task execution, which could be leveraged in future extensions of this work.

– Overpartitioning of a data-parallel computation into more than one task/kernel call leads to finer granularity, which can enable automated hybrid CPU-GPU computing but also incurs increased runtime overhead for the management of the additional tasks.

– Kernel fusion is an optimization for accelerator computations that tries to merge fine-grained tasks/kernel calls into fewer, coarser-grained ones.
– Persistent kernels on GPU are applicable to scenarios with many subsequent kernel calls of one or few statically known types, and can significantly reduce the accumulated kernel latencies for small GPU tasks.
– Operand transfer fusion is a granularity coarsening optimization for the communication of kernel operand data between main memory and accelerator memory.

Overpartitioning. Task-based computations can be generated from higher-level parallel programming models. As an example, we could consider the skeleton programming framework *SkePU* (www.ida.liu.se/labs/pelab/skepu) for GPU-based systems [17,18]. SkePU provides for each supported skeleton (map, reduce, stencil etc.) multiple back-ends (target-specific implementations), e.g. for single-threaded CPU execution, multithreaded CPU execution using OpenMP, and GPU execution in CUDA or OpenCL. Moreover, SkePU also provides a back-end that generates tasks for the StarPU runtime system [14]. From a single skeleton call, a user-defined number of asynchronously executed tasks can be generated, by partitioning the work and thus converting some of the skeleton call's data parallelism into task parallelism. Such "overpartitioning" automatically exploits hybrid CPU-GPU computing via StarPU's dynamic heterogeneous task scheduler [32] at the expense of increased runtime overhead for the management of the additional tasks.

Kernel Fusion. Kernel fusion is an agglomeration optimization for accelerator computations that merges multiple kernels resp. kernel calls into a single one. The purpose of this coarsening of the granularity of accelerator usage is to either improve data locality, or to reduce kernel startup overhead, or to improve the overall throughput by combining memory-bound with arithmetics-bound kernels. Kernel fusion is a special case of the classical *loop fusion* transformation, namely, for the case of parallel loops executing on an accelerator with many parallel hardware threads, such as a GPU.

Kernel fusion can be done in two different ways: *parallel fusion* (by co-scheduling of independent kernels) or *serial fusion* (by serialization of possibly dependent kernels), see also Fig. 9 for illustration.

Serial fusion is particularly effective if it can internalize inter-kernel flow of bulk operand data (i.e., intermediate (sub-)vectors or -matrices) between producer and consumer kernels, and moves the time points of production and consumption of each such data element much closer to each other. Hence, these data elements can now be stored and reused in registers or fast on-chip memory, which reduces the amount of slow off-chip memory accesses and thus increases the arithmetic intensity of the code.

In contrast, parallel fusion does not change the arithmetic intensity of the code, but eliminates kernel startup time overhead, improves thread occupancy and thus utilization of the accelerator especially for kernels with relatively small operands. Moreover, it can lead to overall improved throughput by co-scheduling
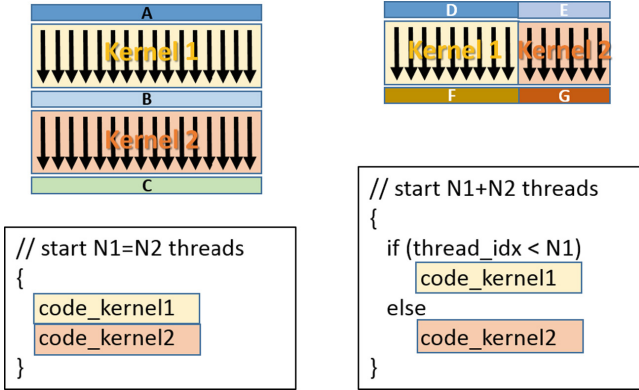
Fig. 9. Left: Serial kernel fusion by sequencing code from (calls to) different kernels in the same parallel loop, preserving per-element data flow dependencies between kernels in the fused code.—Right: Parallel kernel fusion by co-scheduling two previously independent kernel executions within the same "superkernel". Adapted from [59].

memory-bound with arithmetics-bound kernels [60]. For GPUs, parallel fusion can be done at the granularity of individual threads or of thread blocks, the latter of which should give better performance [60].

A number of static kernel fusion techniques especially for compilers targeting GPUs have been presented in the literature, e.g. by Wang *et al.* [59], Wahib and Maruyama [58] and Filipovic *et al.* [20]. Filipovic and Benkner [19] evaluate the effectiveness of parallel kernel fusion on GPU, Xeon Phi and CPU. Wen *et al.* [60] apply parallel kernel fusion in a just-in-time compiler that tries to pair memory-bound with arithmetics-bound kernels. Qiao *et al.* [48] study serial kernel fusion for image processing DSLs.

Persistent Kernel. For scenarios with many small tasks that all execute the same (or just a few different) statically known code, using a *persistent kernel* [29] is another technique to reduce the GPU kernel start-up overhead time (which is, for current CUDA GPUs, in the order of several microseconds, thus significant for small tasks). In contrast to starting a new kernel execution for each GPU task that is supplied with all its input data at its start and that delivers all output data on exit, a persistent kernel is started just once in the beginning and continuously runs on the GPU until it is eventually terminated by the CPU. When idle, the persistent kernel performs busy waiting on its input data buffers until it finds new data to work on, i.e. after it was written (transferred) there by the CPU. It then performs the corresponding operation and writes the data to the corresponding output buffer. The CPU can finally terminate the kernel by writing a special "poison pill" value into an input field that the GPU kernel polls regularly during busy waiting. For example, Maghazeh *et al.* [41] describe how the persistent-kernel technique was used in a packet processing application in telecommunications.

Operand Transfer Fusion. Heterogeneous systems that expose a (physically) distributed memory architecture to the low-level programmer require the explicit memory allocation and transfer of not yet uploaded kernel operand data from main memory to accelerator memory before kernel execution and the transfer of the kernel's output operands back to main memory (or possibly to other accelerator memories) if needed there for subsequent computations. Accelerator APIs provide functions for memory allocation and transfer of operands, such as `cudaMalloc` and `cudaMemcpy`, respectively.

The data transfer time for a bulk operand (e.g., a (sub-)vector or -matrix) of N elements can generally be modeled by a linear cost function $t_{comm} = \alpha + \beta N$, which is characterized by the transfer startup time α and the word transfer time β. On PCIe 3.0-attached GPUs the startup time α can be in an order of about $10\mu s$, with $\alpha/\beta \approx 10^4$ floats [36]. For tasks with small operands, the transfer startup time is thus a none-negligible overhead. Likewise, there is a significant overhead for device memory allocation where required.

A key observation is that multiple operands that can be stored adjacently in both main memory and accelerator memory can be transferred in a single, larger message, thus saving transfer startups compared to separate transfers for each operand. Likewise, device memory can be allocated for such operands by a single call to `cudaMalloc`.

Li and Kessler [36] present a dynamic optimization based on lazy allocation. They replace the standard API functions for lazy execution operand memory allocation and operand transfer by lazy-execution variants that defer their effect until kernel call execution time. At the kernel call, the operands and their (non-) availability in accelerator memory (hence the need for allocation and transfer) are definitely known, even in cases where static analysis could not resolve this information, e.g. due to variable aliasing or statically unknown task mapping. Then, operands to be transferred together will be allocated *consecutively* in memory if possible. This greedy optimization applies to one kernel call at a time.

5.6   High-Level Macro-dataflow Coordination

A common characteristic of the task-parallel programming frameworks discussed so far is that they, despite all abstractions from concrete hardware, do require a considerable expertise in parallel programming to get things right and even more such expertise to get things efficient. One reason is that they intertwine two different aspects of program execution: algorithmic behaviour, i.e., what is to be computed, and organization of task-parallel execution, i.e., how a computation is performed on multiple execution units, including the necessary problem decomposition, communication and synchronization requirements.

The aim of coordination programming is precisely to separate application-centric code from organization-centric code. The term goes back to the seminal work of Gelernter and Carriero [22], but has seen many variations since. For example, S-Net [27] is a declarative coordination language whose design thoroughly avoids the intertwining of computational and organizational aspects.

S-Net achieves a near complete separation of the concern of writing sequential application building blocks (i.e., *application engineering*) from the concern of composing these building blocks to form a parallel application (i.e., *concurrency engineering*).

S-Net defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. We deliberately restrict S-Net to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes*, to conventional programming languages.

An S-Net box is connected to the outside world by two typed streams, a single input stream and a single output stream. The operational behaviour of a box is characterized by a stream transformer function that maps a single data item from the input stream to a (possibly empty) stream of data items on the output stream. S-Net effectively promotes functions implemented in a standard programming language into asynchronously executed stream-processing components.

In order to facilitate dynamic reconfiguration of networks, a box has no internal state, and any access to external state (e.g. file system, environment variables, etc.) is confined to using the streaming network. This allows us to cheaply migrate boxes between computing resources and even having individual boxes process multiple data items concurrently. Boxes execute fully asynchronously: as soon as data is available on the input stream, a box may start computing and producing data on the output stream. Boxes usually represent non-trivial units of computation instead of basic operations as in the original data-flow approach. Hence, S-Net effectively implements a macro data flow model.

It is a distinguishing feature of S-Net that it neither introduces streams as explicit objects nor that it defines network connectivity through explicit wiring. Instead, it uses algebraic formulae to describe streaming networks. The restriction of boxes to a single input and a single output stream (SISO) is essential for this. S-Net provides five network combinators: serial and parallel composition, serial and parallel replication as well as feedback. Any combinator preserves the SISO property: any network, regardless of its complexity, again is a SISO entity.

To summarize, S-Net is an abstract notation to express concurrency in application programs in an abstract and intuitive way. It avoids the typical annoyances of machine-level concurrent programming. Instead, S-Net borrows the idea of streaming networks of asynchronous, stateless components, which segregates applications into their natural building blocks and exposes the data flow between them. We have developed a highly tuned run-time system customized to the specific needs of S-Net [23]. In addition we have developed Distributed S-Net for cluster architectures [26].

S-Net is not at all confined to classical streaming applications as we have demonstrated through a number of numerical application case studies [28,44,45]. We have not yet implemented any of the methods for electromagnetic scattering problems described earlier in this paper, and, unfortunately, for the time being we lack the resources to do so. However, the closest matching algorithm we do

have implemented with S-Net is Tiled Cholesky Factorization, another hierarchical matrix algorithm [10]. Here, S-Net compared very favourably against yet another established task-parallel approach: Intel's Concurrent Collections (CnC) [9,11,33]. In fact, S-Net outperformed CnC both with respect to code size and ease of programming as well as performance and scalability [65].

An interesting question for future work is whether or not—or better to what extent—we may be able to re-produce these positive results for the not dissimilar algorithms discussed in this paper.

6   Summary and Conclusions

In this chapter, we have discussed the properties of hierarchical matrix algorithms arising in electromagnetic scattering problems, and how to parallelize these problems on multicore, heterogeneous, and distributed hardware architectures.

Two different classes of algorithms were discussed in more detail, MLFMA and NESA algorithms. The main difference between these from a parallelization perspective is that in the former, the work performed for groups at different levels varies significantly, while in the latter, the work size per group is uniform. Because of this, a fine-grained parallelization of MLFMA needs to be more intrusive, since the work in coarse level groups needs to be split over threads/ processes.

Both the data structures and the interaction patterns in the hierarchical matrix algorithms are irregular, which is why we suggest to use a parallel programming model that supports asynchronous execution. A pilot implementation using a task parallel programming model for shared memory architectures showed promising results regarding the potential to mix the computational phases during the execution and regarding the resulting utilization of the hardware. A challenging aspect was the relatively small work sizes for individual groups. We discuss different approaches to managing task granularity that could be implemented in future projects.

When working with industrial, or academic, legacy codes, several potentially conflicting interests influence the choices. To change which algorithm is used is typically a major investment, since it is unlikely that this part is well separated from the rest of the code. If the software was started from scratch today, perhaps other algorithmic choices would be made in light of the current prevailing hardware architectures. To achieve the best possible performance probably requires some refactoring of the code, while minimizing the changes to the existing code is relevant both from a cost perspective and a maintainability perspective. Finally, when using high-level programming models which build on some particular implementation of a run-time system, external dependencies are introduced that complicate the administration of the software, and introduce a risk of future incompatibility or discontinuation.

In this chapter we have tried to shed light on some of these choices, to support further work in the area.

References

1. Agullo, E., Aumage, O., Bramas, B., Coulaud, O., Pitoiset, S.: Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. IEEE Trans. Parallel Distrib. Syst. **28**(10), 2794–2807 (2017). https://doi.org/10.1109/TPDS.2017.2697857

2. Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., Takahashi, T.: Task-based FMM for multicore architectures. SIAM J. Sci. Comput. **36**(1), C66–C93 (2014). https://doi.org/10.1137/130915662

3. Anderson, E., et al.: LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)

4. Atkinson, P., McIntosh-Smith, S.: On the performance of parallel tasking runtimes for an irregular fast multipole method application. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 92–106. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_7

5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exper. **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631

6. Benson, A.R., Poulson, J., Tran, K., Engquist, B., Ying, L.: A parallel directional fast multipole method. SIAM J. Sci. Comput. **36**(4), C335–C352 (2014). https://doi.org/10.1137/130945569

7. Bordage, C.: Parallelization on heterogeneous multicore and multi-GPU systems of the fast multipole method for the Helmholtz equation using a runtime system. In: Omatu, S., Nguyen, T. (eds.) Proceedings of the Sixth International Conference on Advanced Engineering Computing and Applications in Sciences, pp. 90–95. International Academy, Research, and Industry Association (IARIA), Curran Associates Inc., Red Hook (2012)

8. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.J.: PaRSEC: exploiting heterogeneity to enhance scalability. Comput. Sci. Eng. **15**(6), 36–45 (2013)

9. Budimlić, Z., Chandramowlishwaran, A., Knobe, K., Lowney, G., Sarkar, V., Treggiari, L.: Multicore implementations of the Concurrent Collections programming model. In: 14th Workshop on Compilers for Parallel Computing, Zürich, Switzerland (2009)

10. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput. **35**(1), 38–53 (2009)

11. Chandramowlishwaran, A., Knobe, K., Vuduc, R.: Performance evaluation of Concurrent Collections on high-performance multicore computing systems. In: 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, USA, pp. 1–12. IEEE, April 2010

12. Cruz, F.A., Knepley, M.G., Barba, L.A.: PetFMM–a dynamically load-balancing parallel fast multipole library. Int. J. Numer. Methods Eng. **85**(4), 403–428 (2011). https://doi.org/10.1002/nme.2972

13. Darve, E., Cecka, C., Takahashi, T.: The fast multipole method on parallel clusters, multicore processors, and graphics processing units. Comptes Rendus Mécanique **339**(2), 185–193 (2011). https://doi.org/10.1016/j.crme.2010.12.005

14. Dastgeer, U., Kessler, C., Thibault, S.: Flexible runtime support for efficient skeleton programming on hybrid systems. In: Proceedings of the ParCo-2011 International Conference on Parallel Computing, Ghent, Belgium, September 2011. Advances in Parallel Computing, vol. 22, pp. 159–166. IOS press (2012). https://doi.org/10.3233/978-1-61499-041-3-159

15. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Proces. Lett. **21**(02), 173–193 (2011)

16. Efield®. http://www.efieldsolutions.com/

17. Enmyren, J., Kessler, C.: SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the 4th Internatioanl Workshop on High-Level Parallel Programming and Applications (HLPP-2010). ACM, September 2010. https://doi.org/10.1145/1863482.1863487

18. Ernstsson, A., Li, L., Kessler, C.: SkePU 2: flexible and type-safe skeleton programming for heterogeneous parallel systems. Int. J. Parallel Program. **46**(1) (2018). https://doi.org/10.1007/s10766-017-0490-5

19. Filipovic, J., Benkner, S.: OpenCL kernel fusion for GPU, Xeon Phi and CPU. In: Proceedings of the 27th International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD 2015), pp. 98–105. IEEE (2015). https://doi.org/10.1109/SAC-PAD.2015.29

20. Filipovic, J., Madzin, M., Fousek, J., Matyska, L.: Optimizing CUDA code by kernel fusion: application on BLAS. J. Supercomput. **71**, 3934–3957 (2015). https://doi.org/10.1007/s11227-015-1483-z

21. Fukuda, K., Matsuda, M., Maruyama, N., Yokota, R., Taura, K., Matsuoka, S.: Tapas: an implicitly parallel programming framework for hierarchical n-body algorithms. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), pp. 1100–1109, December 2016. https://doi.org/10.1109/ICPADS.2016.0145

22. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35**(2), 97–107 (1992)

23. Gijsbers, B., Grelck, C.: An efficient scalable runtime system for macro data flow processing using S-Net. Int. J. Parallel Program. **42**(6), 988–1011 (2014). https://doi.org/10.1007/s10766-013-0271-8

24. Gouin, F.: Methodology for image processing algorithms mapping on massively parallel architectures. Technical report, MINES ParisTech (2018)

25. Gouin, F., Ancourt, C., Guettier, C.: An up to date mapping methodology for GPUs. In: 20th Workshop on Compilers for Parallel Computing (CPC 2018), Dublin, Ireland, April 2018. https://hal-mines-paristech.archives-ouvertes.fr/hal-01759238

26. Grelck, C., Julku, J., Penczek, F.: Distributed S-Net: cluster and grid computing without the hassle. In: 12th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid 2012), Ottawa, Canada. IEEE Computer Society (2012). https://doi.org/10.1109/CCGrid.2012.140

27. Grelck, C., Scholz, S., Shafarenko, A.: Asynchronous stream processing with S-Net. Int. J. Parallel Program. **38**(1), 38–67 (2010). https://doi.org/10.1007/s10766-009-0121-x

28. Grelck, C., Scholz, S.B., Shafarenko, A.: Coordinating data parallel SAC programs with S-Net. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, California, USA. IEEE Computer Society Press, Los Alamitos (2007). https://doi.org/10.1109/IPDPS.2007.370408

29. Gupta, K., Stuart, J.A., Owens, J.D.: A study of persistent threads style GPU programming for GPGPU workloads. In: Innovative Parallel Computing - Foundations and Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012), pp. 1–14. IEEE, May 2012. https://doi.org/10.1109/InPar.2012.6339596
30. Gürel, L., Ergül, O.: Hierarchical parallelization of the multilevel fast multipole algorithm (MLFMA). Proc. IEEE **101**(2), 332–341 (2013). https://doi.org/10.1109/JPROC.2012.2222331
31. Holm, M., Engblom, S., Goude, A., Holmgren, S.: Dynamic autotuning of adaptive fast multipole methods on hybrid multicore CPU and GPU systems. SIAM J. Sci. Comput. **36**(4) (2014). https://doi.org/10.1137/130943595
32. Kessler, C., et al.: Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In: Proceedings of the DATE-2012 Conference on Design, Automation and Test in Europe, pp. 1403–1408. IEEE, March 2012. https://doi.org/10.1109/DATE.2012.6176582
33. Knobe, K.: Ease of use with Concurrent Collections (CnC). In: USENIX Workshop on Hot Topics in Parallelism (HotPar 2009), Berkeley USA (2009)
34. Kurzak, J., Pettitt, B.M.: Massively parallel implementation of a fast multipole method for distributed memory machines. J. Parallel Distrib. Comput. **65**(7), 870–881 (2005). https://doi.org/10.1016/j.jpdc.2005.02.001
35. Lashuk, I., et al.: A massively parallel adaptive fast multipole method on heterogeneous architectures. Commun. ACM **55**(5), 101–109 (2012). https://doi.org/10.1145/2160718.2160740
36. Li, L., Kessler, C.: Lazy allocation and transfer fusion optimization for GPU-based heterogeneous systems. In: Proceedings of the Euromicro PDP-2018 International Conference on Parallel, Distributed, and Network-Based Processing, pp. 311–315. IEEE, March 2018. https://doi.org/10.1109/PDP2018.2018.00054
37. Li, M., Francavilla, M., Vipiana, F., Vecchi, G., Chen, R.: Nested equivalent source approximation for the modeling of multiscale structures. IEEE Trans. Antennas Propag. **62**(7), 3664–3678 (2014)
38. Li, M., Francavilla, M., Vipiana, F., Vecchi, G., Fan, Z., Chen, R.: A doubly hierarchical MoM for high-fidelity modeling of multiscale structures. IEEE Trans. Electromagn. Compat. **56**(5), 1103–1111 (2014)
39. Li, M., Francavilla, M.A., Chen, R., Vecchi, G.: Wideband fast kernel-independent modeling of large multiscale structures via nested equivalent source approximation. IEEE Trans. Antennas Propag. **63**(5), 2122–2134 (2015). https://doi.org/10.1109/TAP.2015.2402297
40. Ltaief, H., Yokota, R.: Data-driven execution of fast multipole methods. Concurr. Comput.: Pract. Exp. **26**(11), 1935–1946 (2014). https://doi.org/10.1002/cpe.3132
41. Maghazeh, A., Bordoloi, U.D., Dastgeer, U., Andrei, A., Eles, P., Peng, Z.: Latency-aware packet processing on CPU-GPU heterogeneous systems. In: Proceedings of the Design Automation Conference (DAC), pp. 41:1–41:6. ACM (2017). https://doi.org/10.1145/3061639.3062269
42. Mautz, J.R., Harrington, R.F.: Electromagnetic scattering from homogeneous material body of revolution. Arch. Electron. Übertragungstech **33**, 71–80 (1979)
43. Nilsson, M.: Fast numerical techniques for electromagnetic problems in frequency domain. Ph.D. thesis, Division of Scientific Computing, Department of Information Technology, Uppsala University (2003)
44. Penczek, F., Cheng, W., Grelck, C., Kirner, R., Scheuermann, B., Shafarenko, A.: A data-flow based coordination approach to concurrent software engineering. In: 2nd Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2012), Minneapolis, USA. IEEE (2012). https://doi.org/10.1109/DFM.2012.14

45. Penczek, F., et al.: Parallel signal processing with S-Net. Procedia Comput. Sci. **1**(1), 2079–2088 (2010). https://doi.org/10.1016/j.procs.2010.04.233. http://www.sciencedirect.com/science/article/B9865-506HM1Y-88/2/87fcf1cee7899f0eeaadc90bd0d56cd3, iCCS 2010
46. Pérez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In: Proceedings of the 2008 IEEE International Conference on Cluster Computing, Tsukuba, Japan, 29 September–1 October 2008, pp. 142–151 (2008). https://doi.org/10.1109/CLUSTR.2008.4663765
47. Puma-EM. https://sourceforge.net/projects/puma-em/
48. Qiao, B., Reiche, O., Hannig, F., Teich, J.: Automatic kernel fusion for image processing DSLs. In: Proceedings of the 21th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2018). ACM, May 2018. https://doi.org/10.1145/3207719.3207723
49. Rao, S., Wilton, D., Glisson, A.: Electromagnetic scattering by surfaces of arbitrary shape. IEEE Trans. Antennas Propag. **30**(3), 409–418 (1982)
50. Seo, S.M., Lee, J.F.: A fast IE-FFT algorithm for solving PEC scattering problems. IEEE Trans. Magn. **41**(5), 1476–1479 (2005)
51. Song, J., Lu, C.C., Chew, W.C.: Multilevel fast multipole algorithm for electromagnetic scattering by large complex objects. IEEE Trans. Antennas Propag. **45**(10), 1488–1493 (1997)
52. Thibault, S.: On Runtime Systems for Task-based Programming on Heterogeneous Platforms. Habilitation à diriger des recherches, L'Université Bordeaux (2018)
53. Thoman, P., Jordan, H., Fahringer, T.: Adaptive granularity control in task parallel programs using multiversioning. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 164–177. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_19
54. Tillenius, M.: SuperGlue: a shared memory framework using data versioning for dependency-aware task-based parallelization. SIAM J. Sci. Comput. **37**(6) (2015). https://doi.org/10.1137/140989716
55. Tillenius, M., Larsson, E., Badia, R.M., Martorell, X.: Resource-aware task scheduling. ACM Trans. Embedded Comput. Syst. **14**(1), 5:1–5:25 (2015). https://doi.org/10.1145/2638554
56. Velamparambil, S., Chew, W.C.: Analysis and performance of a distributed memory multilevel fast multipole algorithm. IEEE Trans. Antennas Propag. **53**(8), 2719–2727 (2005). https://doi.org/10.1109/TAP.2005.851859
57. Vipiana, F., Francavilla, M., Vecchi, G.: EFIE modeling of high-definition multiscale structures. IEEE Trans. Antennas Propag. **58**(7), 2362–2374 (2010)
58. Wahib, M., Maruyama, N.: Scalable kernel fusion for memory-bound GPU applications. In: Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis (SC 2014), pp. 191–202. IEEE (2014). https://doi.org/10.1109/SC.2014.21
59. Wang, G., Lin, Y., Yi, W.: Kernel fusion: an effective method for better power efficiency on multithreaded GPU. In: Proceedings of the IEEE/ACM International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing, pp. 344–350 (2010). https://doi.org/10.1109/GreenCom-CPSCom.2010.102
60. Wen, Y., O'Boyle, M.F., Fensch, C.: MaxPair: enhance OpenCL concurrent kernel execution by weighted maximum matching. In: Proceedings of the GPGPU-11. ACM (2018). https://doi.org/10.1145/3180270.3180272

61. YarKhan, A., Kurzak, J., Dongarra, J.: Quark users' guide: queueing and runtime for kernels. Technical report. ICL-UT-11-02 (2011)
62. Zafari, A.: TaskUniVerse: a task-based unified interface for versatile parallel execution. In: Wyrzykowski, R., Dongarra, J., Deelman, E., Karczewski, K. (eds.) PPAM 2017. LNCS, vol. 10777, pp. 169–184. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78024-5_16
63. Zafari, A., et al.: Task parallel implementation of a solver for electromagnetic scattering problems. CoRR abs/1801.03589 (2018). http://arxiv.org/abs/1801.03589
64. Zafari, A., Larsson, E., Tillenius, M.: DuctTeip: an efficient programming model for distributed task-based parallel computing (2019, submitted)
65. Zaichenkov, P., Gijsbers, B., Grelck, C., Tveretina, O., Shafarenko, A.: The cost and benefits of coordination programming: two case studies in Concurrent Collections (CnC) and S-Net. Parallel Process. Lett. **26**(3) (2016). https://doi.org/10.1142/S0129626416500110
66. Zhang, B.: Asynchronous task scheduling of the fast multipole method using various runtime systems. In: 2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing, pp. 9–16 (2014). https://doi.org/10.1109/DFM.2014.14
67. Zhao, K., Vouvakis, M.N., Lee, J.F.: The adaptive cross approximation algorithm for accelerated method of moments computations of EMC problems. IEEE Trans. Electromagn. Compat. **47**(4), 763–773 (2005)