Task parallel implementation of a solver for electromagnetic scattering problems[☆]

Afshin Zafari[a], Elisabeth Larsson[a,*], Marco Righero[b], M. Alessandro Francavilla[b], Giorgio Giordanengo[b], Francesca Vipiana[c], Giuseppe Vecchi[c]

[a] *Dept. of Information Technology, Uppsala University, Box 337, SE-751 05 Uppsala, Sweden*
[b] *Antenna and EMC Lab (LACE), Istituto Superiore Mario Boella, Torino 10138, Italy*
[c] *Antenna and EMC Lab (LACE), Politecnico di Torino, Torino 10129, Italy*

Abstract

Electromagnetic computations, where the wavelength is small in relation to the geometry of interest, become computationally demanding. In order to manage computations for realistic problems like electromagnetic scattering from aircraft, the use of parallel computing is essential. In this paper, we describe how a solver based on a hierarchical nested equivalent source approximation can be implemented in parallel using a task based programming model. We show that the effort for moving from the serial implementation to a parallel implementation is modest due to the task based programming paradigm, and that the performance achieved on a multicore system is excellent provided that the task size, depending on the method parameters, is large enough.

Keywords: electromagnetics, nested equivalent source approximation, task parallel, low rank approximation, fast multipole method
2010 MSC: 78M15, 78M16, 65Y05

1. Introduction

The Maxwell equations describe the behavior of electromagnetic waves [1]. Researchers have found closed form solutions to the equations only for a few canonical cases. Therefore, when dealing with real-life situations, numerical solutions should be sought. A very useful formulation, often used for scattering

[☆] This article is based upon work from COST Action IC1406 cHiPSET, supported by COST (European Cooperation in Science and Technology). The authors are ordered with respect to affiliation first, and role in the project secondly.

[*] Corresponding author
Email addresses: afshin.zafari@it.uu.se (Afshin Zafari), elisabeth.larsson@it.uu.se (Elisabeth Larsson), righero@ismb.it (Marco Righero), francavilla@ismb.it (M. Alessandro Francavilla), giordanengo@ismb.it (Giorgio Giordanengo), francesca.vipiana@polito.it (Francesca Vipiana), giuseppe.vecchi@polito.it (Giuseppe Vecchi)

and radiation problems, assumes that all the quantities are time harmonic, drops the common time factor, and recasts the problem in terms of equivalent currents, which live on surfaces separating regions composed of homogeneous material or on metallic parts. Expanding the unknown currents in a suitable basis and testing the equations in a weak Galerkin fashion, the problem is then taken into the discrete domain, and translated into a linear system to be solved to determine the unknown current coefficients.

When dealing with real life problems, the matrices involved become too large to be stored and the necessary operations require too much time to be practical. If N is the number of unknowns, usually in the range 10^5–10^7, direct storage of the matrix requires memory resources scaling as $\mathcal{O}(N^2)$, and direct inversion of the matrix requires computational resources scaling as $\mathcal{O}(N^3)$. Different factorizations of the matrices have been proposed in the literature to overcome these difficulties such as the MultiLevel Fast Multipole Algorithm (MLFMA), see, e.g., [2]; FFT-based factorization, see, e.g., [3, 4]; factorizations based on the Adaptive Cross Approximation (ACA), see, e.g., [5]; or based on H2 matrices as the Nested Equivalent Source Approximation (NESA) [6, 7, 8].

All these approximations can be seen as decomposing the matrix of the Galerkin test into a sparse matrix accounting for near field interactions, and a matrix accounting for far field interactions, which is not managed as a full matrix, but in some structured low-complexity, low-memory form. In MLFMA and NESA, much of the gain comes from a multilevel nested approach, where approximations between large scales of the structures are computed using approximations built on smaller scales, in a recursive way.

When computing the entries of the matrix and performing matrix–vector products (MVPs), the advantages of parallel processing can be exploited. In the case of full matrices, the parallelization is trivial: Different threads fill different rows of the matrix and carry out the corresponding multiplications. This approach can also be taken for sparse matrices, even if performance may suffer from a low ratio of computations to memory accesses. When dealing with structured matrices as in MLFMA and NESA, parallelization becomes cumbersome. The very nature of the multilevel approach requires exchange of large volumes of data between levels, making it more difficult to explicitly construct a parallel algorithm due to the irregular data structure and the need for synchronization between levels.

Therefore, we here turn to another type of parallel programming paradigm, where the algorithm is described sequentially and the parallelization is implicit. Task-based parallel programming [9, 10, 11, 12] is one of the successful approaches which simplifies parallel programming by relieving the programmer from thinking about the concurrency control mechanisms in the system architecture.

A task parallel program is written in terms of computational tasks that operate on shared data. Information about the type of accesses (read/write/add) to the data that are performed by each task are provided by the programmer, for example through annotations of the code. The tasks are submitted to a run-time system that dynamically schedules and executes the tasks in parallel,

while respecting all data dependencies, which are automatically extracted from the access annotations. An immediate performance advantage of this approach is that synchronization is applied at the fine-grained task level of the computations instead of using less efficient (implicit or explicit) global synchronization points (barriers).

For the NESA algorithm, the amount of parallelism varies between levels, the components of the matrix representation are of varying sizes, and the data structure also changes with the problem geometry. This makes it complicated and time consuming to try to derive a static parallelization scheme that achieves a reasonable load-balancing across different problems as well as across different computer systems. With a task-based parallel scheme, the available parallelism is automatically extracted by the framework, and the dynamic scheduling allows the execution to adapt to the actual availability of the computational resources. The same task parallel program can run efficiently on different computer systems without modifications.

In this paper, we develop a task parallel implementation of an MVP with a matrix in the NESA format arising from an electrostatic problem in a two-dimensional setting. This pilot implementation is targeted to shared memory multicore systems. The objective is to learn about how the algorithm performs in parallel. This understanding provides the foundation for the future work of developing a task parallel solver for three-dimensional problems running on distributed memory systems. For the three-dimensional case, we can leverage another benefit of task parallel implementations, namely that a task parallel implementation for shared memory with minor changes to the application code can be ported to distributed memory [13, 14].

During the last decade, several different task parallel programming frameworks have been developed based on similar underlying ideas. Some of the main efforts that support both shared and distributed memory systems are OmpSs [9, 15], which employs an OpenMP style of programming with pragmas to define tasks and to annotate accesses, StarPU-MPI [11, 16], which predicts task execution times in order to decide on which type of compute resource CPU/GPU to place a task, and PaRSEC [17], which is extremely fast and scalable, but requires a more involved encoding of the task dependencies.

In this work, we use the shared memory task-based parallel framework SuperGlue [12] developed in the research group of Dr. Larsson at Uppsala University. In the experiments performed in [12], SuperGlue was shown to have lower overhead than related efforts both with respect to small task sizes and large numbers of tasks. For the future work on distributed parallel programming, we will use the DuctTeip framework [14], which is built on top of SuperGlue.

The SuperGlue framework has previously been successfully used for parallelization of a fast adaptive multipole method [18, 19], which is an algorithm that is similar to the NESA algorithm. Another FMM method has been implemented using the StarPU task parallel run-time system [20].

The paper is organized as follows: In Section 2, we briefly recap the basics of the integral equation formulation of electromagnetic scattering and radiation problems, and discuss some efficient algorithms for solving the discretized prob-

lem. In Section 3, we describe the simplified two-dimensional problem that is used for evaluating the parallelization approach. Section 4 provides a general introduction to task based parallel programming, while Section 5 is focused on the implementation of the specific algorithm investigated here. In Section 6, the parallel performance of the new implementation is evaluated, and finally in Section 7, the results are summarized.

2. Integral equation formulation and the nested equivalent source approximation

Let Ω be a volume, whose boundary is $\partial\Omega$, surrounded by a homogeneous medium with wavenumber k and intrinsic impedance η. The electric field $\boldsymbol{e}^s$ at $\boldsymbol{r}$, a point in the exterior medium, radiated by a current $\boldsymbol{j}$ on the surface $\partial\Omega$ is given by

$$\boldsymbol{e}^s(\boldsymbol{j})(\boldsymbol{r}) = -\mathrm{i}\eta k \left(\int_{\partial\Omega} g(\boldsymbol{r},\boldsymbol{r}')\boldsymbol{j}(\boldsymbol{r}')\mathrm{d}\boldsymbol{r}' + \frac{1}{k^2}\nabla \int_{\partial\Omega} g(\boldsymbol{r},\boldsymbol{r}') \div \boldsymbol{j}(\boldsymbol{r}')\mathrm{d}\boldsymbol{r}' \right), \quad (1)$$

where $g(\boldsymbol{r},\boldsymbol{r}') = \exp(-\mathrm{i}k\|\boldsymbol{r}-\boldsymbol{r}'\|)/(4\pi\|\boldsymbol{r}-\boldsymbol{r}'\|)$ is called the Green's function of the exterior medium. If an external electric field $\boldsymbol{e}^i$ impinges on the surface $\partial\Omega$, we can determine the resulting current on $\partial\Omega$ enforcing appropriate boundary conditions on $\partial\Omega$ and, in turn, use this current to evaluate the radiated field in any position. When considering a metallic object, from a numerical viewpoint, we expand the unknown current in terms of an appropriate basis, $\boldsymbol{j}(\boldsymbol{r}) = \sum_{n=1}^{N} j_n \boldsymbol{\phi}_n(\boldsymbol{r})$, and enforce a null-field condition in the weak sense, namely we impose

$$\int_{\partial\Omega} \boldsymbol{e}^s(\boldsymbol{j})(\boldsymbol{r}) \cdot \boldsymbol{\phi}_m(\boldsymbol{r}) = -\int_{\partial\Omega} \boldsymbol{e}^i(\boldsymbol{r}) \cdot \boldsymbol{\phi}_m(\boldsymbol{r}), \quad \forall m = 1,\ldots,N. \quad (2)$$

This results in a linear system $[Z]\,[j] = -\left[e^i\right]$, to be solved to determine coefficients j_n. The entries of the matrix $[Z]$ and the vector $\left[e^i\right]$ are given by

$$Z_{mn} = \int_{\partial\Omega} \boldsymbol{e}^s(\boldsymbol{\phi}_n)(\boldsymbol{r}) \cdot \boldsymbol{\phi}_m(\boldsymbol{r})\mathrm{d}\boldsymbol{r}, \quad e^i_m = -\int_{\partial\Omega} \boldsymbol{e}^i(\boldsymbol{r}) \cdot \boldsymbol{\phi}_m(\boldsymbol{r})\mathrm{d}\boldsymbol{r}. \quad (3)$$

When considering large structures, the resulting linear system is usually solved with an iterative solver, such as BiCGStab or GMRES.

The formulation sketched here is suitable for both scattering and radiation problems. However, it poses three main difficulties, related to the fact that matrix entries are given as convolution integral with a non-local kernel $g(\boldsymbol{r},\boldsymbol{r}')$: Forming the matrix $[Z]$, storing the matrix, and performing the MVPs required by any iterative solver.

As the kernel $g(\boldsymbol{r},\boldsymbol{r}')$ is smooth (when $\boldsymbol{r} \neq \boldsymbol{r}'$) and decreases as $1/\|\boldsymbol{r}-\boldsymbol{r}'\|$, portions of the matrix corresponding to well separated parts of the surface $\partial\Omega$ can be approximated with low-complexity factorizations.

Without going into the details explained in [6], we here sketch the basic idea, for ease of reference. The structure is hierarchically partitioned using an oct-tree. Portions of the matrix $[Z]$ corresponding to interaction between basis functions in near leaf blocks are computed and stored directly, in a sparse matrix $[Z]^{\text{near}}$. Portions of the matrix corresponding to interaction between basis functions in far blocks are computed and stored in an approximate structured way.

We denote with α and β two far groups at the leaf level L, and with $P(\alpha)$ the parent group of the group α. Let $[Z]_{\alpha,\beta}$ denote the block of the matrix $[Z]$ corresponding to the interactions between basis functions in α end β, with size $T \times S$. In low-medium frequency regimes, it is rank deficient [21, 22] and we can approximate it as

$$[Z]_{\alpha,\beta} = [U]_{\alpha} [D]_{\alpha,\beta} [V]_{\beta} \tag{4}$$

where $[U]_{\alpha}$, $[D]_{\alpha,\beta}$, and $[V]_{\beta}$ have sizes $T \times R$, $R \times R$, and $R \times S$, respectively, with $R \ll (T, S)$.

Applying the same idea to a child and its parent group, we see that we can approximate $[D]_{\alpha,\beta}$ as

$$[D]_{\alpha,\beta} = [B]_{\alpha,P(\alpha)} [D]_{P(\alpha)P(\beta)} [C]_{P(\beta),\beta} , \tag{5}$$

so that we have the 1-level approximation

$$[Z]_{\alpha,\beta} = [U]_{\alpha} [B]_{\alpha,P(\alpha)} [D]_{P(\alpha)P(\beta)} [C]_{P(\beta),\beta} [V]_{\beta} . \tag{6}$$

In general, if we move along the family tree induced by the oct-tree division for $\bar{\ell}$ levels, we have

$$\begin{aligned} [Z]_{\alpha,\beta} = [U]_{\alpha} \underbrace{[B]_{\alpha,P(\alpha)} \cdots [B]_{P^{\bar{\ell}-1}(\alpha),P^{\bar{\ell}}(\alpha)}}_{\text{ascending}} [D]_{P^{\bar{\ell}}(\alpha)P^{\bar{\ell}}(\beta)} \\ \underbrace{[C]_{P^{\bar{\ell}}(\beta),P^{\bar{\ell}-1}(\beta)} \cdots [C]_{P(\beta),\beta}}_{\text{descending}} [V]_{\beta} . \end{aligned} \tag{7}$$

Matrices $[U], [D], [V], [B], [C]$ are built enforcing equivalence conditions on fictitious boundaries enclosing the blocks. Their construction is based on standard linear algebra matrix operations and has cost independent of N. From the way they are constructed, we interpret $[U]$ as *receiving matrix*, $[D]$ as *translation matrix*, $[V]$ as *radiation matrix*, and $[B]$ and $[C]$ as *transfer matrices*.

If matrix $[Z]_{\alpha,\beta}$ collects the field due to currents in leaf block β, tested on functions in leaf block α, we ascend the three up to level ℓ_0, so that blocks $P^{L-\ell 0}(\alpha)$ and $P^{L-\ell 0}(\beta)$ are still not touching, and approximate $[Z]_{\alpha,\beta}$ using Equation 7 with $\bar{\ell} = L - \ell_0$.

When dealing with 2D structures, the algorithm can be used with little modifications, due to its purely algebraic nature: The Green's function in Equation 1 becomes $g(\boldsymbol{r}, \boldsymbol{r}') = H_0^{(2)}(k\,|\boldsymbol{r} - \boldsymbol{r}'|)$, with $H_0^{(2)}$ being the Hankel function of the second kind, order zero, and surface integrals become line integrals.

3. The model problem

To test the hypothesis that a task based parallelization is suitable for electromagnetic scattering problems, we implement a simplified two-dimensional problem for a first evaluation, avoiding the complications of implementing the boundary element method, while focusing on the interaction structures when using the hierarchical NESA representation of a matrix.

The simplified problem consists of computing the two-dimensional electric potential $\phi(\boldsymbol{x})$ generated by charges (sources) located at the points $\boldsymbol{x}_i$, $i = 1, \ldots, N$ with charges $q(\boldsymbol{x}_i)$, and evaluating it at the source locations. The dense matrix version of the problem takes the form

$$[\phi] = [K][q], \tag{8}$$

where the matrix elements are given by $k_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j) = -\log(|\boldsymbol{x}_i - \boldsymbol{x}_j|)$, $i \neq j$, and $k_{ii} = 0$, $i, j = 1, \ldots, N$.

In the model problem, the charges are located on a one-dimensional structure within the two-dimensional space. The particular curve that we have used as an example for our source locations is shown in Figure 1. To construct the NESA representation of the matrix, we start by constructing the hierarchical tree structure that should cover the domain of the sources. The tree is constructed with levels $\ell = \ell_0, \ldots, L$. The coarsest level ℓ_0 is chosen such that we have 4 groups along the longest dimension of the domain, see Figure 1. The depth of the tree is determined by the method parameter P. The groups are subdivided as long as the average number of source points in the finest level groups does not fall below P. The example in Figure 1 only shows three levels, but in the examples with $N = 100\,000$ sources we are using in the numerical experiments, there are 8 active levels.

Figure 1 also shows how the computations are divided into near and far field interactions. The near field interactions are computed at the finest level between groups that are close neighbors (left, right, top, bottom, diagonal) and within each group. The near field interaction between groups α and β constitute reduced versions of the global problem (8)

$$[\phi]_\alpha = [\phi]_\alpha + [K]_{\alpha,\beta}[q]_\beta. \tag{9}$$

Far field interactions are computed at each level. The groups that interact in the far field computations are the ones that are not near neighbors at the current level, and whose parents were not part on the interaction on the previous level. In this way, only a limited number of boxes are involved in the far field interaction at each level. If the interactions were computed directly, there would be no computational savings. This is where NESA comes in. Simply put, we compute equivalent charges in each group at each level. We then compute the fields created by the interaction of these charges at the same points, and then transfer the corresponding field to the actual source points. Figure 2 shows the location of the equivalent source points in a parent and child group together with the test surface where the fields are matched in order to form the low rank

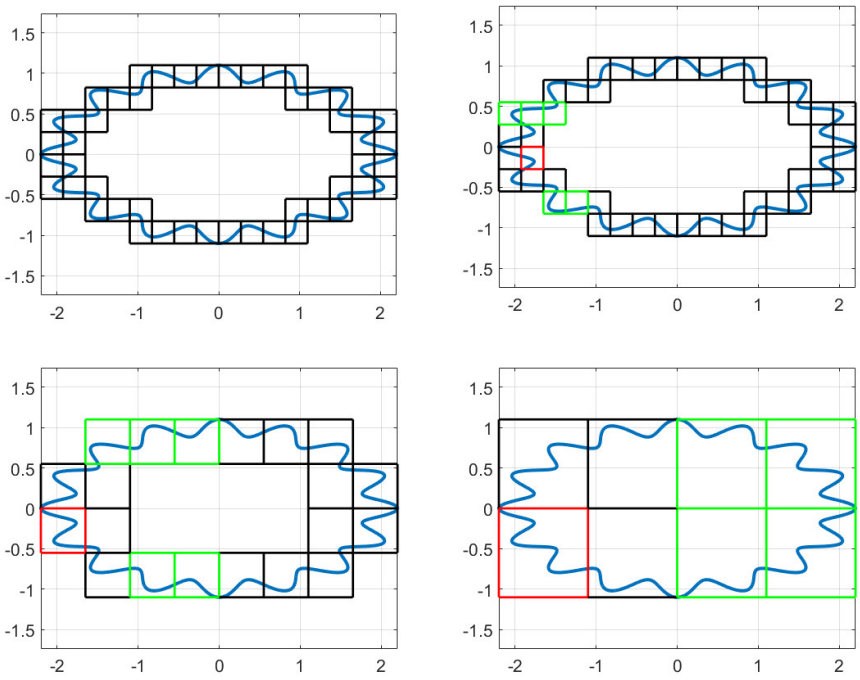

Figure 1: The source points (charges) are located on the blue curve, and the potential is also evaluated at these points. Groups at three different levels are shown. The far field computation contributing to the smallest red group are also shown at each level. The interaction with the whole right half of the domain (marked with green color) is handled at the coarsest level, then additional groups are taken care of at each finer level until only the near field groups remain.

approximation. The number of equivalent charges Q is the same in each group, which is why we can save significantly in the far-field computation.

We will not go into all details here, instead we refer to [6], but we will describe each step in the algorithm in a way that helps the later discussion of the parallel implementation. The far field interaction can be divided into 5 different stages:

Radiation: For each finest level group β, the equivalent sources s at the auxiliary points (see Figure 2) are computed from the actual sources

$$[s]_\beta = [V]_\beta [q]_\beta. \tag{10}$$

Source transfer: Next, equivalent sources are computed for every level of the tree. Each child group β_ℓ at each level ℓ transfers its charge to its parent group $P(\beta_\ell)$

$$[s]_{P(\beta_\ell)} = [s]_{P(\beta_\ell)} + [C]_{P(\beta_\ell),\beta_\ell} [s]_{\beta_\ell}, \quad \ell_0 + 1 \leq \ell \leq L. \tag{11}$$

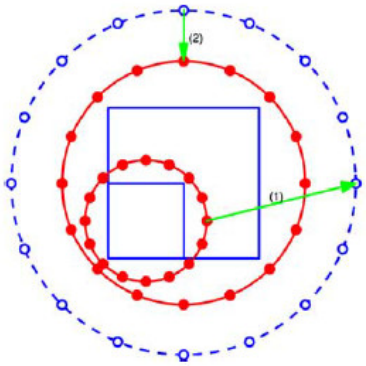

Figure 2: A parent group and one of its children are illustrated. The points on the red circles are where the auxiliary sources are located, and the points on the blue circle are where the potentials are matched.

Translation: For each level ℓ and each (observation) group α_ℓ at that level, the contribution to the potential generated by the groups β_ℓ in the far-field interaction list at the same level is computed

$$[o]_{\alpha_\ell} = [o]_{\alpha_\ell} + [D]_{\alpha_\ell,\beta_\ell}[s]_{\beta_\ell}, \quad \ell_0 \leq \ell \leq L. \tag{12}$$

Potential transfer: The potential contribution at each parent group $P(\alpha_\ell)$ is transferred and added to its child group's potentials.

$$[o]_{\alpha_\ell} = [o]_{\alpha_\ell} + [B]_{\alpha_\ell,P(\alpha_\ell)}[o]_{P(\alpha_\ell)}, \quad \ell_0 + 1 \leq \ell \leq L. \tag{13}$$

Reception: Finally, the potential at the auxiliary points of each finest level group α is transferred to the actual observation points.

$$[\phi]_\alpha = [\phi]_\alpha + [U]_\alpha[o]_\alpha. \tag{14}$$

4. Task parallel programming

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 know how to write a task parallel program and how various aspects of the algorithm are likely to impact performance.

As an example, we consider the shared memory (thread based) parallelization of a dense MVP $y = Ax$. The shared data that the run-time system needs to keep track of in order to ensure a correct end result is the data that will be modified during the execution. In the example, this is only the output vector y. To implement the multiplication as a task parallel algorithm, we need to

break down the operation into smaller components. This can be done in several different ways by blocking or slicing the matrix and vectors into smaller partitions.

Figure 3 shows three common ways of splitting the data structures. A task would correspond to multiplying one slice or block of the matrix with the corresponding part of the vector x. The column-based slicing of the matrix is a bad choice because the whole output vector is touched by each task, and the tasks can therefore not run in parallel (without modifications). The block partitioning as well as the row slicing scheme both allow for parallelism as only a part of the shared data y is touched. The latter two provide the same level of parallelism, but if the multiplication is part of a larger scheme where there is a possibility to interleave tasks from different operations, the block partitioning may be preferred because it provides a larger number of tasks.

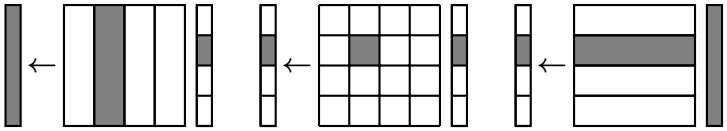

Figure 3: Three different partitionings of the matrix and vectors, and the responsibility of a single task of the MVP. The data accessed by the task is shaded.

The number of tasks, especially in relation to the size of the tasks, is important for performance. With too few tasks, the amount of parallelism is reduced, and there is not enough work for all the threads, which leads to idle time. If there is instead a large number of tasks, but these are very small in terms of computational work, the overhead from managing the tasks in the run-time system becomes large in relation to the task size. Optimally, the task sizes should be chosen such that the task data fits within the local cache.

A slightly different problem that also has a significant impact on performance is bandwidth contention. That is, when all threads are trying to fetch data for their tasks, there may not be enough bandwidth to supply the threads at full speed. Hence, the parallel execution will not be as efficient as theoretically expected. There are different ways to counter bandwidth related problems. If possible, similar computations should be combined such that the number of floating point operations per memory access is increased. This was successfully used, e.g., in [13]. If there is a mix of tasks, with and without bandwidth sensitivity, resource-aware scheduling can improve performance [23]. Finally, accessing contiguous memory locations is more efficient than random memory accesses, which means that data structures should be allocated in such a way that tasks read from contiguous memory as far as possible.

5. Parallel implementation of the fast MVP

In a typical application, the MVP is performed a large number of times within an iterative method while the matrix structure is built once and can

be seen as static. Therefore, we focus on the multiplication algorithm even if the build step could also be parallelized. In the following subsections, we will discuss the algorithm at a general level and the steps taken to convert it to a task parallel implementation.

5.1. The properties of the algorithm

Technically, the hierarchical NESA MVP algorithm consists of a large number of smaller dense MVPs. What is interesting from the parallelization perspective is the number of operations, the size of the operations, and the dependency structure. In order to discuss concrete numbers, we denote the number of active group at each level by N_ℓ. For a full tree in d dimensions $N_{\ell+1} = 2^d N_\ell$, but here the tree is sparse. For the example we are using, the average number of children is around 2.5 leading to $N_{\ell+1} \approx 2.5 N_\ell$. We choose to express all numbers of operation in terms of N_L. Then we have $N_\ell = N_L / 2.5^{(L-\ell)}$.

Going back to the algorithm described in Section 3 by equations (9)–(14), we make the following observations:

- The matrices in the near field operation (9) are on average of size $P \times P$ but the sizes vary between the groups. Each group is involved in at most 3^d near field operations, but in our sparse tree, the average is 5 including the self interaction. The total number of near field tasks is then approximately $5N_L$. These operations can be performed in any order, but updates to the same vector $[s]_\beta$ must be performed one at a time.

- The radiation (10) and reception operations (14) are completely independent. The matrix sizes are on average $Q \times P$ and $P \times Q$, respectively, and the number of operations is N_L for each type.

- The number of source transfer (11) and potential transfer operations (13) are given by the total number of children as each parent and child interact once. We have $N_c = \sum_{\ell=\ell_0+1}^{L} N_\ell \approx N_L \sum_{j=0}^{L-(\ell_0+1)} 2.5^j$ leading to $N_c \leq \frac{5}{3} N_L$. Due to the equivalent source formulation, all the transfer matrices are of the same size, $Q \times Q$. These operations are ordered in the sense that children and parents must complete their tasks in the correct order. Also, when the children are contributing to their parent's equivalent sources, updates by different children must be performed one at a time. Otherwise, operations at the same level are independent.

- The translation operations (12) also involve matrices of size $Q \times Q$. The number of groups in the interaction lists at each level is limited. In our application, the average number is less than 7 for all levels. This means that the number of translation tasks is less than $7 \sum_{\ell=\ell_0}^{L} N_\ell \leq 35/3 N_L$. The translation operations can be performed in any order, but updates to the same location must be performed one at a time.

The hierarchical matrix representation based on groups and levels provides a natural description of the algorithm in terms of tasks. The number of tasks

is large, and even if there are dependencies between the levels in the algorithm, there are plenty of tasks that are independent, and can be interleaved with the constrained tasks. In particular, it should be of benefit to mix the independent near-field and dependent far-field computations.

The sizes of the constituent matrix multiplications are completely determined by the parameters P and Q. However, if we modify P or Q for a fixed problem, we change the total amount of work in the algorithm as well as the memory requirements. By changing Q, we also change the accuracy of NESA. This means that we cannot easily optimize the task sizes. It also means that the number of source points N will not have a strong influence on the performance. For very small N, the number of tasks may be too small to provide work for all threads in parallel, but as soon as N is large enough, the performance will be determined by the task sizes.

For our two-dimensional test problem, the preferred values of P and Q are small with, e.g., $Q = 10$ leading to a tolerance of around $1e - 5$. In the three-dimensional case, when the auxiliary sources are placed on a sphere instead of a circle, the corresponding numbers are larger, for a similar accuracy $Q \approx 100$ is needed [6]. When the tasks are too small, the overhead from managing the tasks dominate and the potential parallel speedup is reduced or lost. The precise sizes needed to achieve speedup will be investigated in section 6.

One way to avoid the overhead resulting from having too small tasks would be to let each task manage several groups. This would reduce the number of tasks, but it would increase the complexity of the algorithm, making it harder to implement.

Another performance issue that the MVP will suffer from is bandwidth contention. All MVPs are somewhat sensitive to bandwidth as the number of floating point operations $\mathcal{O}(N^2)$ are proportional to the number of matrix elements $\mathcal{O}(N^2)$. If the operations are instead transformed into matrix–matrix operations, the situation is improved, as the number of operations become $\mathcal{O}(N^3)$, while the storage is still $\mathcal{O}(N^2)$. In the NESA algorithm this would be possible for the radiation and reception steps, since the transfer matrices between parents and children with the same relative positions are the same. Then all similar products could be combined into one operation. However, this would also make the implementation much more complicated with additional packing and unpacking procedures, which also could be time consuming.

5.2. The task parallel implementation

We have chosen to implement the most natural task based formulation, where each task corresponds to one small MVP. In the sequential code, each of the small matrices are allocated in contiguous memory using a C++ user defined `Matrix` data type, and the MVPs are performed by directly calling the BLAS routine `cblas_dgemv`. The `Matrix` type is used also for the input and output vectors.

The changes that are needed to produce a task parallel code are minor. First we need to protect the output vectors from simultaneous accesses by different

tasks. In SuperGlue, shared data is protected by handles that control accesses to the data. Therefore, we introduce a new type `SGMatrix`, which basically equips a `Matrix` with a handle. The type definition is shown in Appendix A.

Next we need to, unless it is already available, define a SuperGlue task class that provides an MVP. The class is also shown in Appendix A. When the task class is in place, we construct a new `gemv` subroutine that instead of directly calling BLAS, constructs the corresponding task and submits it to the run-time system. The task parallel `gemv` subroutine is shown as Program 1.

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

Program 1: The subroutine that submits an MVP task.

We can now write the whole program in task parallel form, by replacing the data types and the subroutine calls by their counterparts. Program 2 shows how SuperGlue is invoked, how matrices are created and how one MVP is performed.

```
1 #include "superglue.hpp"
2
3 SuperGlue<Options> *sgEngine;
4
5 int main(int argc , char *argv[]){
6   // Start the task parallel run-time system
7   sgEngine = new SuperGlue<Options>(config.cores);
8   // Allocating matrices (filled with 0.0)
9   int P=300, Q=100;
10  Matrix &a = * new Matrix (Q,P,0.0);
11  Matrix &x = * new Matrix (P,1,0.0);
12  Matrix &y = * new Matrix (Q,1,0.0);
13  // Make these protected shared data
14  SGMatrix &A = *new SGMatrix(a);
15  SGMatrix &X = *new SGMatrix(x);
16  SGMatrix &Y = *new SGMatrix(y);
17  // Write the algorithm with calls to the MVP
18  // subroutine. For each new call, the
19  // corresponding task is submitted
20  gemv(A,X,Y);
21  // Wait for all tasks to finish
22  sgEngine->barrier();
23 }
```

Program 2: An example of how SuperGlue is included in a program performing an MVP.

As mentioned above, more details on the task class implementation and the shared data type is given in Appendix A.

6. Performance evaluation

The experiments have been performed on one shared memory node of the Tintin cluster at the Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). Each node is dual socket with two AMD Opteron 6220 (Bulldozer) processors running at 3.0 GHz with 64 GB or 128 GB memory. 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.

The same problem with $N = 100\,000$ source points is solved in all experiments, but the method parameters P (the average number of source points at the finest level) and Q (the number of auxiliary points used for each group) are varied between the experiments.

As discussed in the previous section, the properties of the near and far field parts of the algorithm are quite different. Here, we first evaluate the two parts separately to establish their performance at different parameter choices, and then move to the full MVP algorithm.

For each test case, we show speedup results computed as

$$S_p = \frac{T_1}{T_p}, \tag{15}$$

where T_p is the execution time of the task parallel implementation running on p threads. In the graphs, we also show the speedup of the sequential code compared with the parallel code running on one thread. In the optimal case they would be close to equal, but for problems with small tasks, we can see a slight difference.

We also show execution traces, where each task is shown as a triangle with its base at the starting time of the task and the tip at the end of that task's execution. Traces provide a good way of visualizing the execution, the scheduling, and potential performance issues.

Finally, we provide tables with detailed information on execution times, speedup and utilization. We define the utilization as

$$U_p = \frac{T_p^t}{T_p}, \tag{16}$$

where T_p^t is the fraction of the execution time spent executing tasks. The remaining time represents the overhead of managing tasks, idle time when threads are waiting for tasks, and load imbalance at the end of the execution.

6.1. The far field computation

Figure 4 shows how the speedup of the far field computations varies with Q for two different values of P. The speedup grows with Q as the task sizes increase, and it also grows with P, which affects the size of the radiation and

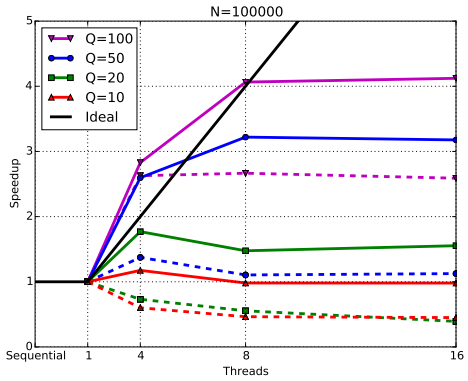

Figure 4: Speedup for different values of Q for the far field computation with $P = 400$ (solid lines) and with $P = 50$ (dashed lines).

reception tasks. However increasing only Q is not enough, which tells us that transfer and translation tasks do not scale very well.

To learn more about the details of the execution, we look at the traces for execution on 16 threads shown in Figure 5. For the problem with small task sizes there are several things to observe. First, the colors that represent different types of tasks are not mixed. This tells us that the tasks are so small that they finish executing before more tasks have been submitted. In fact, only 15 threads are visible in the trace because one thread is constantly occupied with task submission. The small size of the tasks also leads to idle time in between tasks and the resulting execution is not efficient.

For the problem with larger P and Q, we see the benefit of having the larger radiation and reception tasks. These are completely independent and provide enough work to occupy all threads, and the troublesome transfer tasks are nicely embedded within these computations. The translation tasks are smaller, but also more independent than the transfer tasks, and they are mostly scheduled densely within the trace. The task submission is visible also here, but it only occupies the first half of the execution time for thread 0.

Looking at the second trace, we might expect a near optimal speedup as the schedule has very little idle time. However, the speedup was never above 4 in any of the experiments. To investigate if bandwidth contention is involved, we have computed the average execution time for each type of task for different numbers of threads. The result is shown in Figure 6. In the left subfigure, we can see that all of the small tasks experience a slowdown or longer execution time per task on more threads. The larger tasks fare better, and it also seems that tasks with transposed matrices perform better. With an increase of 4 in the individual task execution times as in the worst case here, it is not possible to get an overall speedup higher than 4 on 16 threads. We can also here find the explanation to why the speedup is larger than the theoretical best for 4 threads. When only one thread is used it has dedicated use of the FPU. When instead each FPU is occupied by two threads, the computations become relatively slower, and the

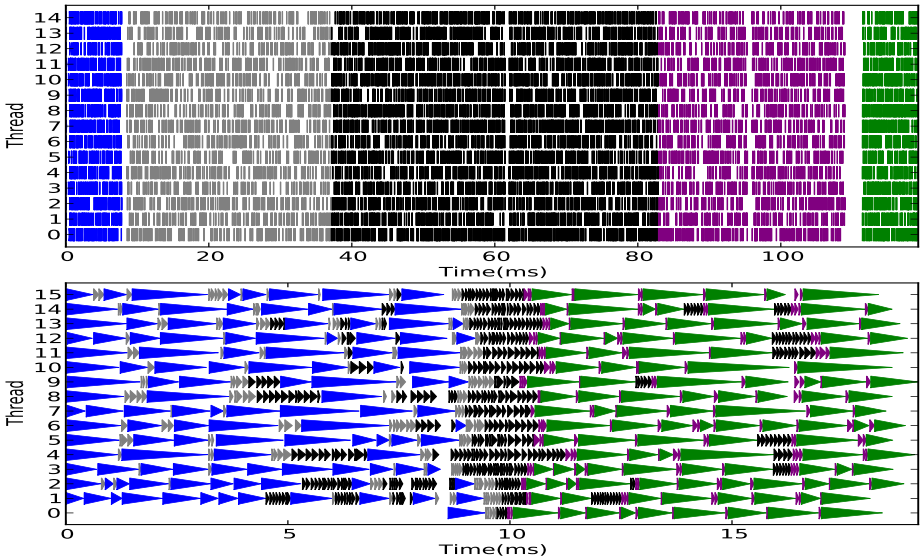

Figure 5: Execution traces for the far field computation for $Q = 10$ and $P = 50$ (top) and for $Q = 100$ and $P = 400$ (bottom).

bandwidth issue is somewhat improved. In the right subfigure, where the tasks are larger, the slowdown is reduced, but there is still contention.

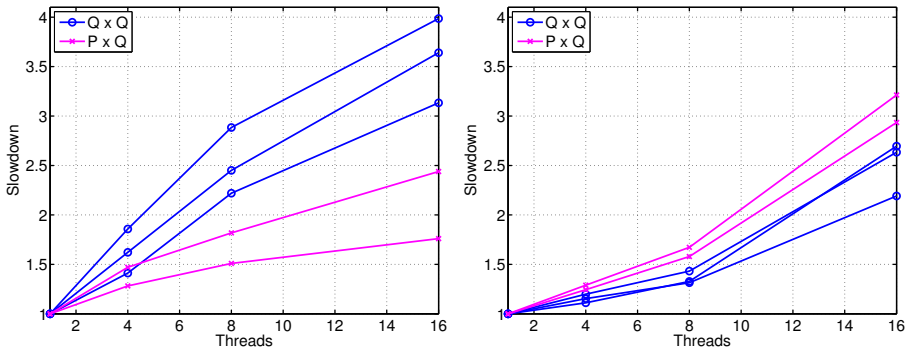

Figure 6: Increase in individual task execution times due to resource contention for $P = 50$, $Q = 10$ (left) and $P = 400$, $Q = 100$ (right).

Finally, Table 1 shows speedup and utilization for the far field computations. For the problem with small task sizes, both speedup and utilization are very low, while for the problem with larger sizes, the utilization is very good, but due to the slowdown of individual tasks, the maximum speedup stays around 4.

Table 1: Performance results for the far field computation for two different parameter sets. T_p is the execution time for p threads, S_p is the speedup, S_p^* is the theoretical optimal speedup, and U_p is the utilization.

$Q = 10,\ P = 50$				
p	T_p [ms]	S_p	S_p/S_p^*	U_p
1	27	1	1.00	0.38
4	86	0.31	0.16	0.04
8	113	0.24	0.06	0.02
16	119	0.23	0.03	0.01
$Q = 100,\ P = 400$				
1	95	1	1.00	0.98
4	31	3.0	1.52	0.92
8	20	4.8	1.19	0.91
16	19	5.0	0.62	0.90

6.2. The near field computation

For the near field computations, the picture is quite different. Speedup results are shown in Figure 9. Also here, the speedup increases with P, but instead of leveling out at 4, it is superoptimal and lands at 10 for $P = 400$.

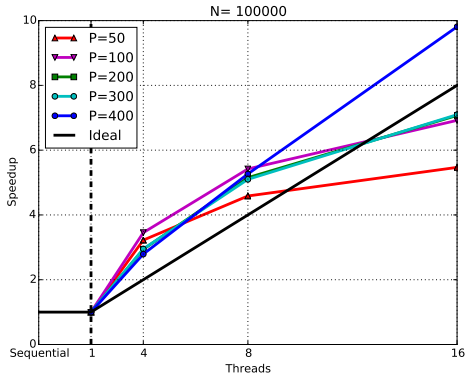

Figure 7: Speedup for different values of P for the near field computation.

The traces in Figure 8 really show the strength of task parallel programming. The tasks are of highly varying sizes, depending on the number of source points in each group, but are scheduled densely across all threads. For the larger value of P, the task submission phase is not visible in the trace.

Table 2 shows performance results for the near field computation. Already for the smaller P both speedup and utilization are high, and for the larger P, the performance is excellent.

6.3. The complete MVP

Here we have run the complete MVP allowing the near field and far field tasks to mix when possible. Figure 9 shows the speedup results for the complete

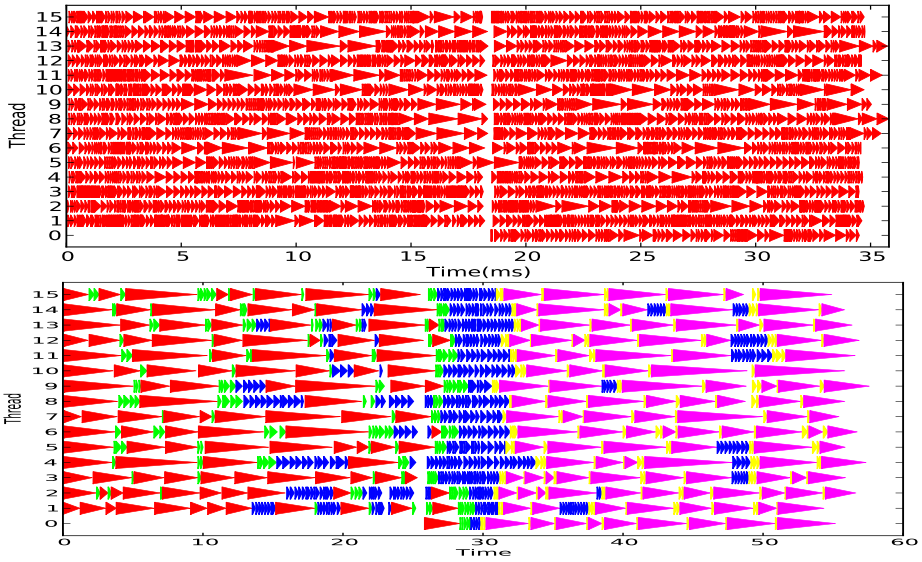

Figure 8: Execution traces for the near field computation for $P = 50$ (top) and for $P = 400$ (bottom).

runs. Here we have chosen to use $P = 300$ for the problem with larger task sizes, even though the previous results showed that $P = 400$ is more efficient. The reason is that we wanted the far field computation to be a larger part of the execution trace than for the $P = 400$ case. For the problem with small tasks, the speedup is around 2, which is in between the far field result 0.23 and the near field result 6.2. For the problem with large tasks, the speedup 7.3 is very close to the result for the near field 7.45, and the far field result of 4.0 does not seem to impact the speedup at all.

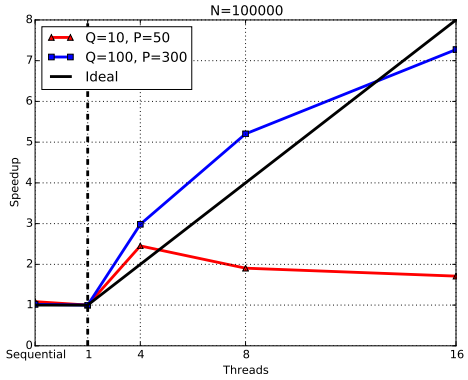

Figure 9: Speedup for two combinations of P and Q resulting in different task sizes.

To get further information, we consider the traces in Figure 10. In the

Table 2: Detailed performance results for the near field computation for two different parameter sets. T_p is the execution time for p threads, S_p is the speedup, S_p^* is the theoretical optimal speedup, and U_p is the utilization.

$P = 50$				
p	T_p [ms]	S_p	S_p/S_p^*	U_p
1	222	1	1.00	0.97
4	66	3.4	1.69	0.91
8	42	5.2	1.30	0.89
16	36	6.2	0.77	0.87
$P = 400$				
1	3848	1	1.00	1.00
4	1379	2.8	1.40	0.98
8	732	5.3	1.31	0.95
16	398	9.7	1.21	0.94

case with small tasks, there is no mixing, and the speedup will clearly be a combination of that for the individual cases. However, when the tasks are larger, the executions are mixed and the far field computation is efficiently executed within the near field computation.

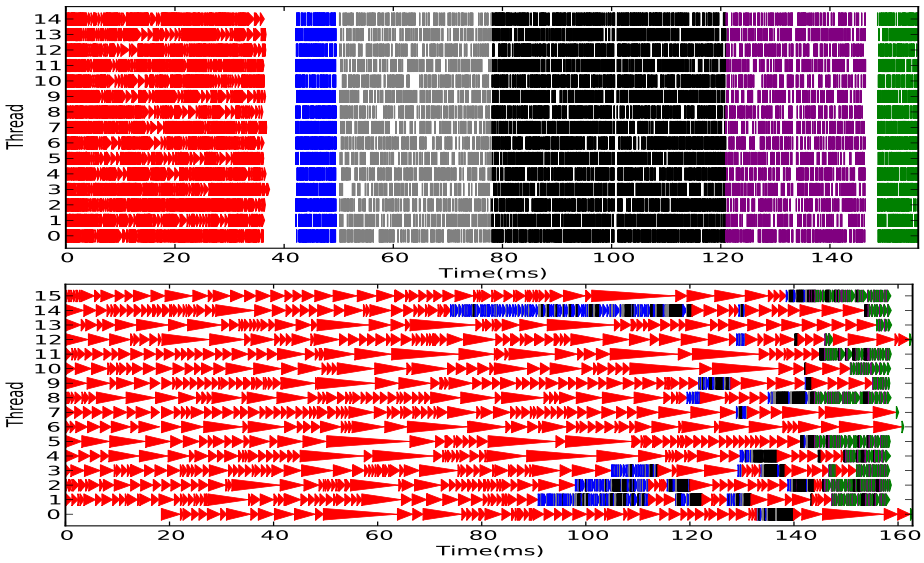

Figure 10: Execution traces for the whole computation for $Q = 10$ and $P = 50$ (top) and for $Q = 100$ and $P = 300$ (bottom).

To further understand the improved performance, we also look at the slow down profiles for the tasks. Figure 11 shows that the slowdown is significant, especially for the smallest tasks, for the execution trace where the tasks do not mix. However, for the mixed trace with larger tasks, the performance is perfect

Table 3: Performance results for the complete computation for two different parameter sets. T_p is the execution time for p threads, S_p is the speedup, S_p^* is the theoretical optimal speedup, and U_p is the utilization.

$Q = 10,\ P = 50$				
p	T_p [ms]	S_p	S_p/S_p^*	U_p
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

for all tasks. A slowdown factor of 2 would be expected due to the shared FPUs. This can be understood in the following way: The near field tasks performed well already from the start. The far field tasks did not perform as well, but this was for the case when they were executed in parallel on all threads. Here, there are fewer far field tasks running in parallel at each point in time, and the performance degradation is avoided.

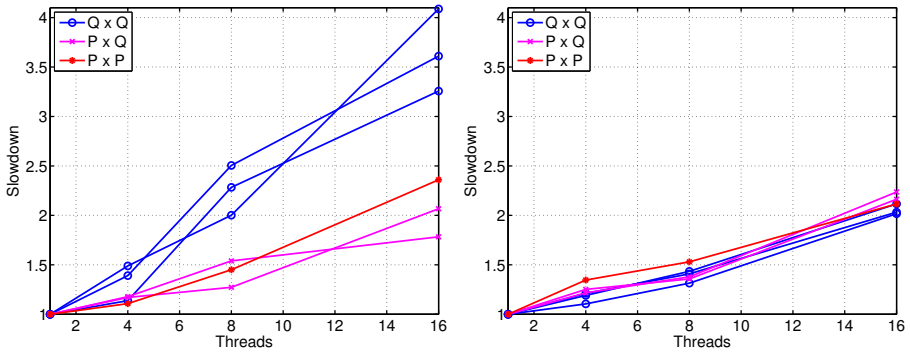

Figure 11: Increase in individual task execution times for the complete execution for $P = 50$, $Q = 10$ (left) and for $P = 300$, $Q = 100$ (right).

Table 3 shows the speedup and utilization results for the whole execution, confirming that the results are excellent if the task sizes are large enough.

7. Summary

The most challenging aspect of the NESA algorithm from a task parallel point of view is not the hierarchical dependencies as might be expected, but rather the small task sizes. For sizes typical for a two-dimensional problem, the

tasks are too small to provide scaling to more than a few threads. However, for sizes appropriate for a three-dimensional problem, the performance is close to optimal and a significant speedup is achieved.

The task parallel programming model provided an easy way to implement a parallel code with very small changes to the original implementation. Moreover, the asynchronous nature of the task execution allowed for mixing of the different types of tasks, which proved to be the key to achieve high performance for the complete MVP including the hierarchical far field computation.

The conclusion is that it is possible to achieve excellent performance on shared memory systems for the NESA type of MVPs provided the task sizes can be made large enough, and that the task based approach is promising for the development of a distributed three-dimensional solver.

Acknowledgments

This article is based upon work from COST Action IC1406 High-Performance Modelling and Simulation for Big Data Applications (cHiPSET), supported by COST (European Cooperation in Science and Technology). The computations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) under Project p2009014.

Appendix A. Shared data types and tasks

Program 3 shows the definition of the C++ class `SGMatrix`, which equips a `Matrix` with a handle such that it can be used as a protected shared data in a task parallel execution.

```
1 #include "superglue.hpp"
2
3 extern SuperGlue *sgEngine;
4
5 class SGMatrix
6 {
7   Handle<Options> *sg_handle;
8   Matrix *M;
9   bool trans;
10 public:
11   SGMatrix(Matrix &m){
12     sg_handle = new Handle<Options>;
13     trans=false;
14     M = &m;
15   }
16   // Further methods omitted here for brevity
17 };
```

Program 3: The SGMatrix data type, which protects a shared matrix with a handle.

Program 4 shows the `SGTaskGemv` SuperGlue task class that provides the MVP task.

```
1  class SGTaskGemv : public Task<Options,3>{
2  private:
3    SGMatrix *A,*x,*y;
4  public:
5    bool transA;
6    enum{COL_MAJOR,ROW_MAJOR};
7    SGTaskGemv(SGMatrix &A_,SGMatrix &x_,SGMatrix &y_)
8    {
9      A = &A_;
10     x = &x_;
11     y = &y_;
12     register_args();
13   }
14   void register_args(){
15     Handle<Options> &hA = A->get_handle();
16     Handle<Options> &hx = x->get_handle();
17     Handle<Options> &hy = y->get_handle();
18     register_access(ReadWriteAdd::read, hA);
19     register_access(ReadWriteAdd::read, hx);
20     register_access(ReadWriteAdd::add , hy);
21     transA = false;
22   }
23   void run(){
24     int M = A->get_matrix()->rows();
25     int N = A->get_matrix()->cols();
26     double *Mat= A->get_matrix()->get_data_memory();
27     double *X  = x->get_matrix()->get_data_memory();
28     double *Y  = y->get_matrix()->get_data_memory();
29     cblas_dgemv(COL_MAJOR,transA,M,N, 1.0, Mat, M,
30                 X, 1, 1.0, Y, 1);
31   }
32 };
```

Program 4: The MVP task class.

In the task constructor, lines 7–13, the data is copied into the task, and the access type for each handle is registered. In the NESA algorithm, the input vector and the matrix are constant, so the registration of the read accesses for A and v could have been omitted. The commutative *add* access type is used for the output vector to allow reordering of the accesses by different tasks. The alternative would be a *write* access, which implies that tasks that access the

same vector y need to be executed in the same order as they are submitted, resulting in less flexibility and potential performance losses [23]. When a task is executed by the SuperGlue run-time system, the `run` method of the task is called.

[1] J. R. Mautz, R. F. Harrington, Electromagnetic scattering from homogeneous material body of revolution, Arch. Elektron. Übertragungstech. 33 (1979) 71–80.

[2] J. Song, C.-C. Lu, W. C. Chew, Multilevel fast multipole algorithm for electromagnetic scattering by large complex objects, IEEE Trans. Antennas Propag. 45 (10) (1997) 1488–1493. doi:10.1109/8.633855.

[3] S. M. Seo, J.-F. Lee, A fast IE-FFT algorithm for solving PEC scattering problems, IEEE Trans. Magn. 41 (5) (2005) 1476–1479. doi:10.1109/TMAG.2005.844564.

[4] F. Vipiana, M. Francavilla, G. Vecchi, EFIE modeling of high-definition multiscale structures, IEEE Trans. Antennas Propag. 58 (7) (2010) 2362–2374. doi:10.1109/TAP.2010.2048855.

[5] K. Zhao, M. N. Vouvakis, J.-F. Lee, The adaptive cross approximation algorithm for accelerated method of moments computations of EMC problems, IEEE Trans. Electromagn. Compat. 47 (4) (2005) 763–773. doi:10.1109/TEMC.2005.857898.

[6] M. Li, M. Francavilla, F. Vipiana, G. Vecchi, R. Chen, Nested equivalent source approximation for the modeling of multiscale structures, IEEE Trans. Antennas Propag. 62 (7) (2014) 3664–3678. doi:10.1109/TAP.2014.2321139.

[7] M. Li, M. Francavilla, F. Vipiana, G. Vecchi, Z. Fan, R. Chen, A doubly hierarchical MoM for high-fidelity modeling of multiscale structures, IEEE Trans. Electromagn. Compat. 56 (5) (2014) 1103–1111. doi:10.1109/TEMC.2014.2306691.

[8] M. Li, M. A. Francavilla, R. Chen, G. Vecchi, Wideband fast kernel-independent modeling of large multiscale structures via nested equivalent source approximation, IEEE Trans. Antennas Propag. 63 (5) (2015) 2122–2134. doi:10.1109/TAP.2015.2402297.

[9] J. M. Pérez, R. M. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan, 2008, pp. 142–151. doi:10.1109/CLUSTR.2008.4663765.

[10] J. Dongarra, P. Luszczek, PLASMA, in: Encyclopedia of Parallel Computing, Springer US, Boston, MA, 2011, pp. 1568–1570. doi:10.1007/978-0-387-09766-4_153.

[11] C. Augonnet, S. Thibault, R. Namyst, P. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency Computat.: Pract. Exper. 23 (2) (2011) 187–198. doi:10.1002/cpe.1631.

[12] M. Tillenius, SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization, SIAM J. Scientific Computing 37 (6). doi:10.1137/140989716.

[13] M. Tillenius, E. Larsson, E. Lehto, N. Flyer, A scalable RBF-FD method for atmospheric flow, J. Comput. Phys. 298 (2015) 406–422. doi:10.1016/j.jcp.2015.06.003.

[14] A. Zafari, E. Larsson, M. Tillenius, DuctTeip: A task-based parallel programming framework for distributed memory architectures, Tech. Rep. 2016-010, Department of Information Technology, Uppsala University (Jun. 2016).

[15] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, J. Labarta, ClusterSs: a task-based programming model for clusters, in: Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8–11, 2011, pp. 267–268. doi:10.1145/1996130.1996168.

[16] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, S. Thibault, StarPU-MPI: Task programming over clusters of machines enhanced with accelerators, in: Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23–26, 2012, pp. 298–299. doi:10.1007/978-3-642-33518-1_40.

[17] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, J. J. Dongarra, PaRSEC: Exploiting heterogeneity to enhance scalability, Comput. Sci. Eng. 15 (6) (2013) 36–45. doi:10.1109/MCSE.2013.98.

[18] S. Engblom, On well-separated sets and fast multipole methods, Appl. Numer. Math. 61 (10) (2011) 1096–1102. doi:10.1016/j.apnum.2011.06.011.

[19] M. Holm, S. Engblom, A. Goude, S. Holmgren, Dynamic autotuning of adaptive fast multipole methods on hybrid multicore CPU and GPU systems, SIAM J. Scientific Computing 36 (4). doi:10.1137/130943595.

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

[21] S. Kapur, D. E. Long, IES3: a fast integral equation solver for efficient 3-dimensional extraction, in: Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1997, San Jose, CA, USA, November 9-13, 1997, 1997, pp. 448–455. doi:10.1109/ICCAD.1997.643574.

[22] M. Bebendorf, Approximation of boundary element matrices, Numer. Math. 86 (4) (2000) 565–589. doi:10.1007/PL00005410.

[23] M. Tillenius, E. Larsson, R. M. Badia, X. Martorell, Resource-aware task scheduling, ACM Trans. Embedded Comput. Syst. 14 (1) (2015) 5:1–5:25. doi:10.1145/2638554.