Scalable Parallelization Strategies to Accelerate NuFFT Data Translation on Multicores

Yuanrui Zhang[1], Jun Liu[1], Emre Kultursay[1],
Mahmut Kandemir[1], Nikos Pitsianis[2,3], and Xiaobai Sun[3]

[1] Pennsylvania State University, University Park, USA
{yuazhang,jxl1036,euk139,kandemir}@cse.psu.edu
[2] Aristotle University,Thessaloniki, Greece
[3] Duke University, Durham, U.S.A.
{nikos,xiaobai}@cs.duke.edu

Abstract. The non-uniform FFT (NuFFT) has been widely used in many applications. In this paper, we propose two new scalable parallelization strategies to accelerate the data translation step of the NuFFT on multicore machines. Both schemes employ geometric tiling and binning to exploit data locality, and use recursive partitioning and scheduling with dynamic task allocation to achieve load balancing. The experimental results collected from a commercial multicore machine show that, with the help of our parallelization strategies, the data translation step is no longer the bottleneck in the NuFFT computation, even for large data set sizes, with any input sample distribution.

1 Introduction

The non-uniform FFT (NuFFT) [2] [7] [15] has been widely used in many applications, including synthetic radar imaging [16], medical imaging [13], telecommunications [19], and geoscience and seismic analysis [6]. Unlike the Fast Fourier Transform (FFT) [4], it allows the sampling in the data or frequency space (or both) to be unequally-spaced or non-equispaced. To achieve the same $O(N \log N)$ computational complexity as the FFT, the NuFFT translates the unequally-spaced samples to the equally-spaced points, and then applies the FFT to the translated Cartesian grid, where the complexity of the first step, named *data translation* or *re-sampling*, is linear to the size of the sample ensemble.

Despite the lower arithmetic complexity compared to the FFT, the data translation step has been found to be the most time-consuming part in computing NuFFT [18]. The reason lies in its irregular data access pattern, which significantly deteriorates the memory performance in modern parallel architectures [5]. Furthermore, as data translation is essentially a matrix-vector multiplication with an irregular and sparse matrix, its intrinsic parallelism cannot be readily exploited by conventional compiler based techniques [17] that work well mostly for regular dense matrices. Many existing NuFFT algorithms [2] [8] [14] [12] try to reduce the complexity of data translation while maintaining desirable accuracy through mathematical methods, e.g., by designing different kernel

P. D'Ambra, M. Guarracino, and D. Talia (Eds.): Euro-Par 2010, Part II, LNCS 6272, pp. 125–136, 2010.

functions. In a complementary effort, we attempt to improve the performance of data translation through different parallelization strategies that take into account the architectural features of the target platform, without compromising accuracy.

In our previous work [20], we developed a tool that automatically generates a fast parallel NuFFT data translation code for user-specified multicore architecture and algorithmic parameters. This tool consists of two major components. The first one applies an architecture-aware parallelization strategy to input samples, rearranging them in off-chip memory or data file. The second one instantiates a parallel C code based on the derived parallel partitions and schedules, using a pool of codelets for various kernel functions. The key to the success of this tool is its parallelization strategy, which directly dictates the performance of the output code. The scheme we developed in [20] has generated significant improvements for the data translation computation, compared to a more straightforward approach, which does not consider the architectural features of the underlying parallel platform. However, it is limited to small data set sizes, e.g., $2K \times 2K$, with non-uniformly distributed samples; the parallelization takes excessively long time to finish when the data size is large. This slow-down is mainly due to the use of recursive geometric tiling and binning during parallelization, which is intended for improving the data locality (cache behavior) of the translation code.

To overcome this drawback, in this paper, we design and experimentally evaluate two new scalable parallelization strategies that employ an equally-sized tiling and binning to cluster the unequally-spaced samples, for both uniform and non-uniform distributions. The first strategy is called the *source driven parallelization*, and the second one is referred to as the *target driven parallelization*. Both strategies use dynamic task allocation to achieve load balancing, instead of the static approach employed in [20]. To guarantee the mutual exclusion in data updates during concurrent computation, the first scheme applies a special parallel scheduling, whereas the second one employs a customized parallel partitioning. Although both schemes have comparable performance for the data translation with uniformly distributed sources, the target driven parallelization outperforms the other when using the input with non-uniformly distributed sources, especially on a large number of cores, in which case synchronization overheads become significant in the first scheme. We conducted experiments on a commercial multicore machine, and compared the execution time of the data translation step with the FFT from FFTW [11]. The collected results demonstrate that, with the help of our proposed parallelization strategies, the data translation step is no longer the primary bottleneck in the NuFFT computation, even for non-uniformly distributed samples with large data set sizes.

The rest of the paper is organized as follows. Section 2 explains the NuFFT data translation algorithm and its basic operations. Section 3 describes our proposed parallelization strategies in detail. Section 4 presents the results from our experimental analysis, and finally, Section 5 gives the concluding remarks.

2 Background

2.1 Data Translation Algorithm

Data translation algorithms vary in the type of data sets they target and the type of re-gridding or re-sampling methods they employ. In this paper, we focus primarily on convolution-based schemes for data translation, as represented by Eq.(1) below, where the *source* samples S are unequally-spaced, e.g., in the frequency domain, and the *target* samples T are equally-spaced, e.g., in an image domain. The dual case with equally-spaced sources and unequally-spaced targets can be treated in a symmetrical way.

$$\mathbf{v}(T) = \mathbf{C}(T, S) \cdot \mathbf{q}(S). \tag{1}$$

In Eq.(1), $\mathbf{q}(S)$ and $\mathbf{v}(T)$ denote input source values and translated target values, respectively, and $\mathbf{C}$ represents the convolution kernel function. The set S for source locations can be provided in different ways. In one case, the sample coordinates are expressed and generated by closed-form formulas, as in the case with the sampling on a polar grid (see Figure 1 (a)). Alternatively, the coordinates can be provided in a data file as a sequence of coordinate tuples, generated from a random sampling (see Figure 1 (b)). The range and space intervals of the target Cartesian grid T are specified by the algorithm designer, and they can be simplified into a single *oversampling factor* [7] since the samples are uniformly distributed. With an oversampling factor of α, the relationship between the number of sources and targets can be expressed as $\mid T \mid = \alpha \mid S \mid$.

The convolution kernel $\mathbf{C}$ is obtained either by closed-form formulas or numerically. Examples for the former case are the Gaussian kernel and the central B-splines [14] [7], whereas examples for the latter case are the functions obtained numerically according to the local least-square criterion [14] and the min-max criterion [8]. In each case, we assume that the function evaluation routines are provided. In addition, the kernel function is of local support, i.e., each source is only involved in the convolution computation with targets within a window, and vice versa. Figure 1 (c) shows an example for a single source in the 2D case, where the window has a side length of w.

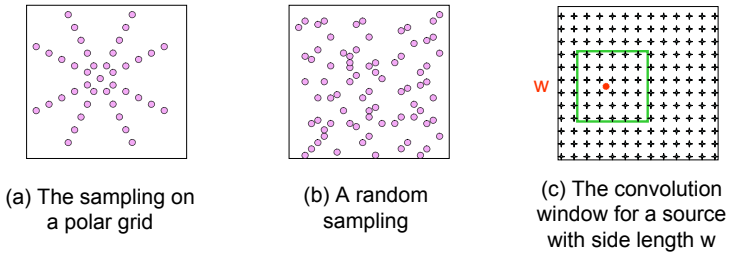


Fig. 1. Illustration of different sampling schemes and local convolution window

2.2 Basic Data Translation Procedure

The above data translation algorithm can be described by the following pseudo code:

```
for each source s_i in S
    for each target t_j in T ⋂ window(s_i)
        V(t_j) += c(t_j, s_i) × Q(s_i)
```

where Q is a one-dimensional array containing the source values regardless of the geometric dimension, V is a d-dimensional array holding the target values, and c represents a kernel function over $T \times S$, e.g., the Gaussian kernel $e^{(|t_j - s_i|)/\sigma}$ where $\mid t_j - s_i \mid$ denotes the distance between a target and a source. The target coordinates are generated on-demand during the computation based on the source coordinates, the oversampling value (α), and the range of the data space, e.g., $L \times L$ in the 2D case. In the pseudo code, the outer loop iterates over sources, because it is easy to find the specific targets for a source within the window, but not vice versa. We name this type of computation *the source driven computation*. The alternate computation is *the target driven computation*, whose outer loop iterates over targets. The complexity of the code is $\mathrm{O}(w^d \times \mid S \mid)$; however, since w is usually very small (in other words, w^d is near constant), the data translation time is linear with the number of sources $\mid S \mid$.

3 Data Translation Parallelization

3.1 Geometric Tiling and Binning

The convolution-based data translation is essentially a matrix-vector multiplication with sparse and irregular matrices. While the sparsity stems from the local window effect of the kernel function, the irregularity is caused by the unequally-spaced sources. In this case, the conventional tiling on dense and regular matrices [17] cannot help to achieve high data reuse. For instance, the source samples s_1 and s_2 from the same tile, as shown in Figure 2 (a), may update different targets located far away from each other in the data space, as indicated by Figure 2 (b). To exploit target reuse, the *geometric tiling* [3] is employed to cluster the sources into cells/tiles based on their spatial locations. The tiles can be equally-sized (Figure 3 (a)) or unequally-sized (Figure 3 (b)), with the latter obtained through an adaptive tiling based on the sample distribution. In either case, the basic data translation procedure can be expressed as:

```
for each non-empty source tile T_k in S
    for each source s_i in T_k
        for each target t_j in T ⋂ window(s_i)
            V(t_j) += c(t_j, s_i) × Q(s_i)
```

Associated with tiling is a process called *binning*. It reshuffles the source data in the storage space, e.g., external memory or file, according to tiles. In terms of data movements, the complexity of an equally-sized tiling and binning is $O(\mid S \mid)$,

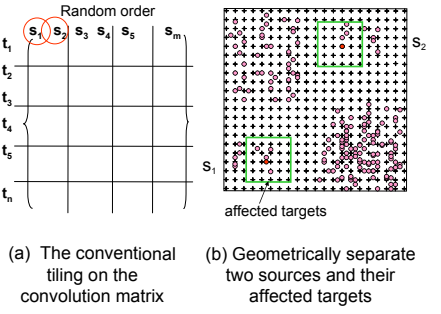


Fig. 2. Conventional tiling

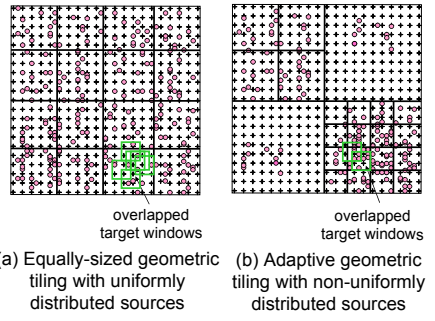


Fig. 3. Geometric tiling

whereas the cost of an adaptive recursive tiling and binning is $O(\mid S \mid \log \mid S \mid)$. The latency of the latter increases dramatically when the data set size is very large, as observed in [20]. Consequently, in this work, we adopt the equally-sized tiling, irrespective of the source distribution, i.e., *uniform distribution* or *non-uniform distribution*, as illustrated in Figure 3. In this way, the non-Cartesian grid of sources is transformed to a Cartesian grid of tiles before any parallel partitioning and scheduling is applied, and this separation makes our parallelization strategies scalable with the data set sizes.

3.2 Parallelization Strategies

To parallelize the data translation step, two critical issues need to be considered: mutual exclusion of data updates and load balancing. On the one hand, a direct parallelization of the source loop of the code in Section 2.2 or the source tile loop of the code in Section 3.1 may lead to incorrect results, as threads on different cores may attempt to update the same target when they are processing geometrically nearby sources concurrently. Although parallelizing the inner target loop can avoid this, it would cause significant synchronization overheads. On the other hand, a simple equally-sized parallel partitioning in the data space may lead to unbalanced workloads across multiple processors when input sources are non-uniformly distributed.

To address these issues, we have designed two parallelization strategies that aim at accelerating data translation on emerging multicore machines with on-chip caches. One of these strategies is called the *source driven parallelization*, and the other is referred to as the *target driven parallelization*. They are intended to be used in the context of the source driven computation. To ensure mutual exclusion of target updates, the first strategy employs a special parallel scheduling, whereas the second strategy applies a customized parallel partitioning. Both the strategies use a recursive approach with dynamic task allocation to achieve load balance across the cores in the target architecture.

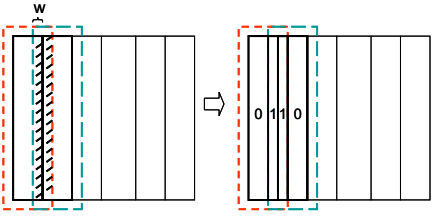


Fig. 4. One-dimensional partition with a 2-step scheduling for the 2D case

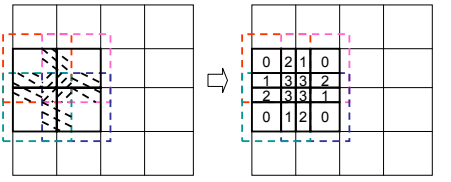


Fig. 5. Two-dimensional partition with a 4-step scheduling for the 2D case

1) Source Driven Parallelization

For a data space containing both sources and targets, an explicit partitioning of sources will induce an implicit partitioning of targets, and vice versa, because of the local convolution window effect. The source driven parallelization carries out parallel partitioning and scheduling in the source domain, whereas the other operates on the targets. In both cases, the source domain has been transformed into a Cartesian grid of tiles through geometric tiling and binning (inside each tile, the samples are still unequally-spaced).

When partitioning the sources, a special scheduling is indispensable to guaranteeing the mutual exclusion of target updates. Consider Figure 4 and Figure 5 for example. No matter how the sources in a 2D region are partitioned, e.g., using one-dimensional or two-dimensional partition, adjacent blocks cannot be processed at the same time due to potentially affected overlapping targets, as indicated by the dashed lines. However, if further dividing the neighboring source regions into smaller blocks according to the target overlapping patterns, a scheduling can be found to process those adjacent source regions through several steps, where at each step, the sub-blocks with the same number (indicated in the figures) can be executed concurrently. And, a synchronization takes place as moving from one step to another. Our observation is that a one-dimensional partition needs a 2-step scheduling to eliminate the contention, whereas a two-dimensional partition needs 4 steps. In general, an x-dimensional partition ($1 \leq x \leq d$) requires a 2^x-step scheduling to ensure correctness.

Based on this observation, our source driven parallelization scheme is designed as follows. Given m threads and a d-dimensional data space, first factorize m into $p_1 \times p_2 \times \ldots \times p_d$, and then divide dimension i into $2p_i$ segments ($1 \leq i \leq d$), which results in $2p_1 \times 2p_2 \times \ldots 2p_d$ blocks in the data space, and finally schedule every 2^d neighboring blocks using the same execution-order pattern. Figure 6 (a) illustrates an example for the 2D case with $m = 16$ and $p_1 = p_2 = 4$. The blocks having the same time stamp (number) can be processed concurrently, provided that the window side length w is less than any side length of a block. In the case where m is very large and this window size condition no longer holds, some threads will be dropped to decrease m, until the condition is met. Although a similar d'-dimensional partition ($d' < d$) with a $2^{d'}$-step scheduling can also be used for a d-dimensional data space, e.g., a one-dimensional partition for the 2D

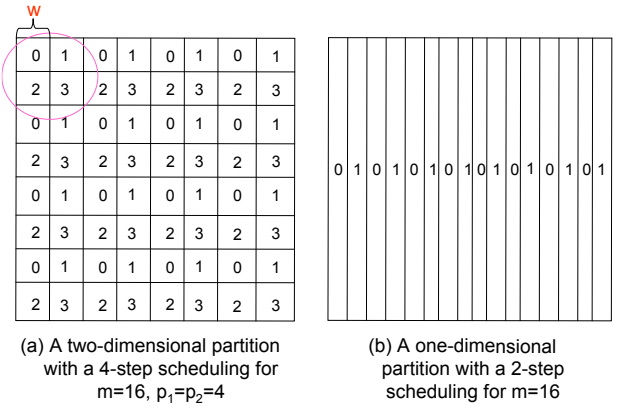


Fig. 6. Illustration of non-recursive source driven parallelization in the 2D case

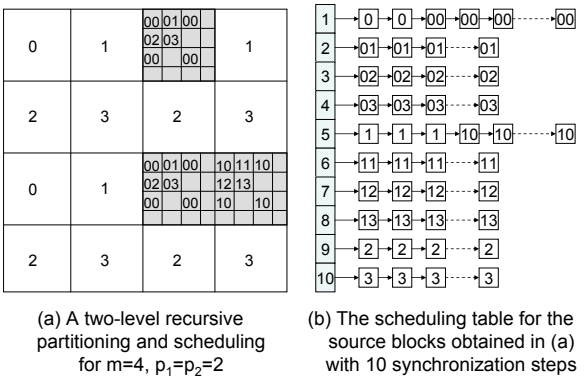


Fig. 7. Illustration of recursive source driven parallelization in the 2D case

case, as shown in Figure 6 (b), it is not as scalable as the d-dimensional partition when m is huge.

The scheme explained so far works well with uniformly distributed sources, but not with non-uniformly distributed ones, as it can cause unbalanced workloads in the latter case. However, a slight modification can fix this problem. Specifically, since the amount of computation in a block is proportional to the number of sources it contains, the above scheme can be recursively applied to the source space until the number of sources in a block is less than a preset value (δ). The recursion needs to ensure that the window size condition is not violated. Figure 7 (a) depicts an example of two-level recursive partitioning for the 2D case, where $m = 4$, $p_1 = p_2 = 2$, and the shaded regions are assumed to have dense samples. The corresponding concurrent schedule is represented by the table shown in Figure 7 (b). The synchronization takes place every time after processing each group of blocks pointed by a table entry. However, within each

group, there is no processing order for the blocks, which will be dynamically allocated to the threads at run time. The selection of threshold δ has an impact on the recursion depth as well as the synchronization latency. A smaller value of δ usually leads to deeper recursions and higher synchronization overheads; but, it is also expected to have better load balance. Thus, there is a tradeoff between minimizing synchronization overheads and balancing workloads, and careful selection of δ is important for the overall performance.

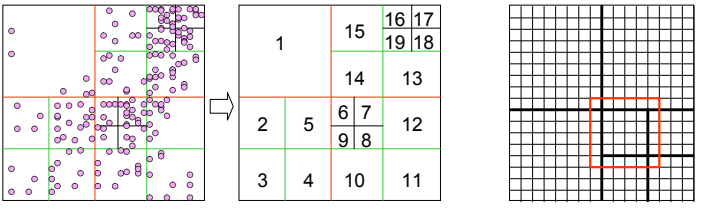
2) Target Driven Parallelization

We have also designed a target driven parallelization strategy to perform recursive partitioning and scheduling in the target domain, which has the advantage of requiring no synchronization. Given m threads and a d-dimensional data space, this scheme employs a 2^d-branch geometric tree to divide the target space into blocks recursively until the number of sources associated with each block is below a threshold (σ), and then uses a neighborhood traversal to obtain an execution order for those blocks, based on which it dynamically allocates their corresponding sources to the threads at run time without any synchronization. Figure 8 (a) shows an example for the 2D case with quadtree partition [9] and its neighborhood traversal. Since target blocks are non-overlapping, there is no data update contention. Although the associated sources of adjacent blocks are overlapping, there is no coherence issue, as sources are only read from the external memory. The neighborhood traversal helps improve data locality during computation through the exploitation of source reuses. The threshold σ is set to be less than $|S|/m$ and greater than the maximum number of sources in a tile. A smaller value of σ usually results in more target blocks and more duplicated source accesses because of overlapping; but, it is also expected to exhibit better load balance, especially with non-uniformly distributed inputs. Therefore, concerning the selection of value for σ, there is a tradeoff between minimizing memory access overheads and balancing workloads.

In addition, to find the associated sources for a particular target block is not as easy as the other way around. Typically, one needs to compare the coordinates of each source with the boundaries of the target block, which will take $O(\mid S \mid)$ time. Our parallelization scheme reduces this time to a constant by aligning the window of a target block with the Cartesian grid of tiles, as depicted in Figure 8 (b). Although this method attaches irrelevant sources to each block, the introduced execution overheads can be reduced by choosing proper tile size.

4 Experimental Evaluation

We implemented these two parallelization strategies in our tool [20], and evaluated them experimentally on a commercial multicore machine. Two sample inputs are used, both of which are generated in a 2D region of 10×10, with a data set size of $15K \times 15K$. One contains random sources that are uniformly distributed in the data space, whereas the other has samples generated on a



(a) An example of quadtree partition in the target domain and neighborhood traversal among target blocks

(b) An illustration of aligned window to the source tiles for a target block

Fig. 8. Example of target driven parallelization in the 2D case. Especially, (b) shows the Cartesian grid of tiles and a target partition with bolded lines, where the window of a target block is aligned to the tiles, indicated by red color.

polar grid with non-uniform distribution. The kernel is the Gaussian function and oversampling (α) is 2.25. The convolution window side length w is set to be 5 in terms of the number of targets affected in each dimension. In this case, each source is involved in the computation with 25 targets, and the total number of targets is $22.5K \times 22.5K$. The underlying platform is Intel Harpertown multicore machine [1], which features a dual quad-core operating at a frequency of 3GHz, 8 private L1 caches of size 32KB and 4 shared L2 caches of size 6MB, each connecting to a pair of cores.

We first investigated the relationship between the tile size and the cache size (both L1 and L2) on the target platform, and analyzed the impact of tile size on the performance of binning (the most time-consuming process in the parallelization phase) and data translation. The tile size is determined based on the cache size so that all the sources and their potentially affected targets in each tile are expected to fit in the cache space. For non-uniform distributions, the tile size is actually calculated based on the assumption of uniform distribution. Figure 9 shows the execution time of binning and data translation for the input with uniformly distributed sources (random samples), using the source driven parallelization strategy. The cache size varies from 16KB to 12MB. The four groups depicted from left to right are the results of our experiments with 1 core, 2 cores, 4 cores, and 8 cores, respectively. For binning, the time decreases as the cache size (or tile size) increases, irrespective of the number of cores used. The reason is that, each tile employs an array data structure to keep track of its sources, when the tile size increases, or equivalently, the number of tiles decreases, it is likely that the data structure of the corresponding tile is in the cache when binning a source, i.e., the cache locality is better. Hence, fewer tiles makes binning faster. In contrast, the data translation becomes slower when the tile size increases, since the geometric locality of the sources in each tile worsens, which in turn reduces the target reuse. When both binning and data translation are concerned, we can see that, there is a lowest point (minimum execution time) in each group, which is around the tile size 1.5 MB, half of L2 cache size per core (3MB) in the Harpertown processor. A similar behavior can also be observed for

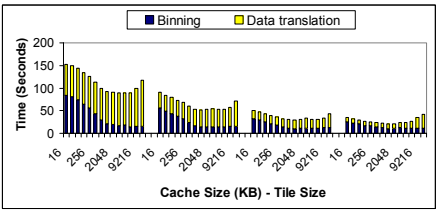


Fig. 9. Execution time of binning and data translation under uniform distribution, as cache (tile) size varies

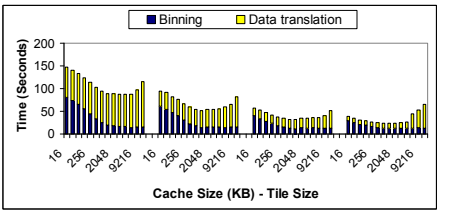


Fig. 10. Execution time of binning and data translation under non-uniform distribution, as cache (tile) size varies

the input with non-uniformly distributed sources (samples on the polar grid), as shown in Figure 10, but with a shifted minimum value.

We then evaluated the efficiency of our two parallelization strategies, using the best tile size found from the first group of experiments. Figure 11 and Figure 12 show their performance with tuned σ and δ, respectively, for non-uniformly distributed sources. We can see that, in both the plots, the execution time of data translation scales well with the number of cores, due to improved data locality; however, the execution time of geometric tiling and binning reduces much slower when the number of cores increases, as this process involves only physical data movements in the memory. In particular, the two execution times become very close to each other on 8 cores. This indicates that our proposed parallelization strategies are suitable for a pipelined NuFFT at this point, where the three steps of the NuFFT, namely, parallelization, data translation and FFT are expected to have similar latencies in order to achieve balanced pipeline stages for streaming applications. Further, the two parallelization strategies have comparable performance for data translation with uniformly distributed sources, as shown by the group of bars on the left in Figure 13; however, the target driven strategy outperforms the other by 13% and 37% on 4 cores and 8 cores, respectively, with non-uniformly distributed sources, as depicted by the group of bars on the right in Figure 13, where the respective data translation times using source and target driven schemes on 4 cores are 23.1 and 20.1 seconds, and on 8 cores are 17.1 and 10.8 seconds. This difference is expected to become more pronounced as the number of cores increases, since there will be more synchronization overheads in the source driven scheme.

We also conducted a performance comparison between the data translation using the target driven parallelization strategy on the input samples, and the FFT obtained from FFTW with "FFTW_MEASURE" option [11] [10] on the translated target points. Figure 14 presents the collected results with non-uniformly distributed sources. The graph shows that the execution times of data translation and FFT are comparable on 1, 2, 4 and 8 cores, respectively. In particular, the data translation becomes faster than the FFT as the number of cores increases. This good performance indicates that, with our parallelization strategies, the data translation step is no longer the bottleneck in the NuFFT computation.

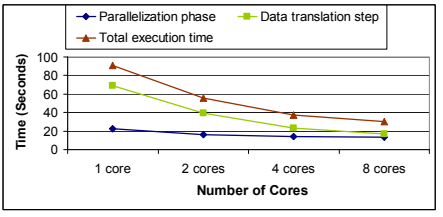


Fig. 11. Performance of the source driven parallelization with non-uniformly distributed sources

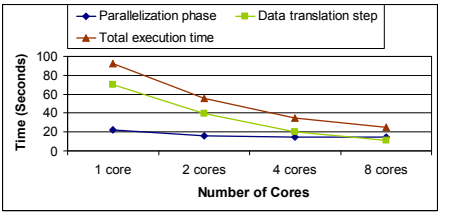


Fig. 12. Performance of the target driven parallelization with non-uniformly distributed sources

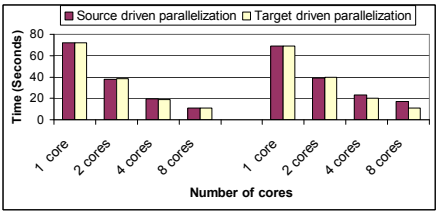


Fig. 13. Data translation time with the two parallelization strategies, for uniform and non-uniform distributions, respectively

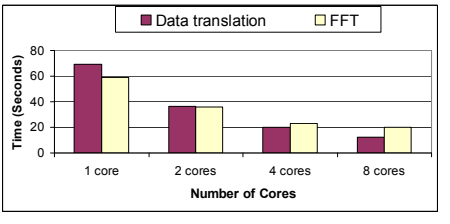


Fig. 14. Execution time of the data translation using target driven parallelization and the FFT from FFTW

5 Concluding Remarks

In this work, we proposed two new parallelization strategies for the NuFFT data translation step. Both schemes employ geometric tiling and binning to exploit data locality, and use recursive partitioning and scheduling with dynamic task allocation to achieve load balance on emerging multicore architectures. To ensure the mutual exclusion in data updates during concurrent computation, the first scheme applies a special parallel scheduling, whereas the second one employs a customized parallel partitioning. Our experimental results show that, the proposed parallelization strategies work well with large data set sizes, even for non-uniformly distributed input samples, which help NuFFT achieve good performance for data translation on multicores.

Acknowledgement

This research is supported in part by NSF grants CNS 0720645, CCF 0811687, OCI 821527, CCF 0702519, CNS 0720749, and a grant from Microsoft.

References

1. `http://www.intel.com`
2. Beylkin, G.: On the fast Fourier transform of functions with singularities. Applied and Computational Harmonic Analysis 2, 363–381 (1995)

3. Chen, G., Xue, L., et al.: Geometric tiling for reducing power consumption in structured matrix operations. In: Proceedings of IEEE International SOC Conference, pp. 113–114 (September 2006)
4. Cooley, J., Tukey, J.: An algorithm for the machine computation of complex Fourier series. Mathematics of Computation 19, 297–301 (1965)
5. Debroy, N., Pitsianis, N., Sun, X.: Accelerating nonuniform fast Fourier transform via reduction in memory access latency. In: SPIE, vol. 7074, p. 707404 (2008)
6. Duijndam, A., Schonewille, M.: Nonunifrom fast Fourier transform. Geophysics 64, 539 (1999)
7. Dutt, A., Rokhlin, V.: Fast Fourier transforms for nonequispaced data. SIAM Journal on Scientific Computing 14, 1368–1393 (1993)
8. Fessler, J.A., Sutton, B.P.: Nonuniform fast Fourier transforms using min-max interpolation. IEEE Transactions on Signal Processing 51, 560–574 (2003)
9. Finkel, R., Bentley, J.: Quad trees: A data structure for retrieval on composite keys. Acta Informatica 4, 1–9 (1974)
10. Frigo, M.: A fast Fourier transform compiler. ACM SIGPLAN Notices 34, 169–180 (1999)
11. Frigo, M., Johnson, S.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing, vol. 3, pp. 1381–1384 (May 1998)
12. Greengard, L., Lee, J.Y.: Accelerating the nonuniform fast Fourier transform. SIAM Review 46, 443–454 (2004)
13. Knopp, T., Kunis, S., Potts, D.: A note on the iterative MRI reconstruction from nonuniform k-space data. International Journal of Biomedical Imaging 6, 4089–4091 (2007)
14. Liu, Q., Nguyen, N.: An accurate algorithm for nonuniform fast Fourier transforms (NUFFTs). IEEE Microwaves and Guided Wave Letters 8, 18–20 (1998)
15. Liu, Q., Tang, X.: Iterative algorithm for nonuniform inverse fast Fourier transform (NU-IFFT). Electronics Letters 34, 1913–1914 (1998)
16. Renganarayana, L., Rajopadhye, S.: An approach to SAR imaging by means of non-uniform FFTs. In: Proceedings of IEEE International Geoscience and Remote Sensing Symposium, vol. 6, pp. 4089–4091 (July 2003)
17. Renganarayana, L., Rajopadhye, S.: A geometric programming framework for optimal multi-level tiling. In: Proceedings of ACM/IEEE Conference on Supercomputing, May 2004, p. 18 (2004)
18. Sorensen, T., Schaeffter, T., Noe, K., Hansen, M.: Accelerating the nonequispaced fast Fourier transform on commodity graphics hardware. IEEE Transactions on Medical Imaging 27, 538–547 (2008)
19. Ying, S., Kuo, J.: Application of two-dimensional nonuniform fast Fourier transform (2-d NuFFT) technique to analysis of shielded microstrip circuits. IEEE Transactions on Microwave Theory and Techniques 53, 993–999 (2005)
20. Zhang, Y., Kandemir, M., Pitsianis, N., Sun, X.: Exploring parallelization strategies for NUFFT data translation. In: Proceedings of Esweek, EMSOFT (2009)