Proceedings of ALGORITMY 2000
Conference on Scientific Computing, pp. 227–235

PARALLEL WAVELET-BASED COMPRESSION OF TWO-DIMENSIONAL DATA

MARIA LUCKA* AND TOR SØREVIK†

Abstract. In this paper we explain our strategy for parallelizing a wavelet based compression routine. The compression routine is designed to work within a earthquake simulation model where huge amount of data is written and read from disks. As the compression routine makes up for a significant part of the computation it is very important that it parallelize well.

We have implemented the parallel version in OpenMP. This makes the implementation easy, but it hides a fundamental problem. The optimal data layout on a distributed memory system changes for the different parts of the algorithm. Thus either data has to be redistributed or the processors have to do remote data access. Our experiments show that the remote data access work pretty well on the Origin system thus for the number of CPU less or equal to 32, no significant slow down, due to remote data access is seen.

Key words. parallel computing, data compression, wavelet transform, OpenMP

AMS subject classifications. 68P30, 65Y05

1. Introduction. Full 3-D finite difference scheme for the governing equations of the ground motion of an earthquake is an extremely compute and memory intensive task. In a recent paper Moczo et. al. [5] introduced a technique called Combined Memory Optimization (CMO), which provides a solution to the data storage problem.

The technique has two main ingredients. First it applies and idea of Graves [4] on how to keep only a limited number of 2-D planes in memory while doing the maximum possible time steps on these data. Only when no more updates are possible the "left" 2-D plane, is written to memory while the next plane to the "right" is read from memory. Repeating this process we get a wave like progressing in time through the data.

While saving memory space, this method might put severe strains on secondary storage and IO-bandwidth. To limit these problems they mixed in a second ingredient, data compression. Before data are sent to disk they are compressed by a wavelet based compression routine.

While the above mentioned paper [5] concentrated on memory optimization the present paper will focus on optimizing for computational speed of the code. Unfortunately it turns out that the compression routine adds a significant overhead to the computational cost. In the worst case we have measured as much 30% extra CPU-use when compression is turned on. Thus while saving storage space the CMO technique increase the need for faster arithmetic.

Our main technique for speeding up this code is parallelization. For explicit finite difference scheme over a regular domain, parallelization is in principal straightforward, though in practice a number of technical details need special care. These details will not be covered in this paper, instead we concentrate on the problems related to parallelization of the compression scheme.

The computational most expensive part of our compression scheme is the Fast Wavelet Transform (the FWT). Parallelization of this has been studied elsewhere.

* Geophysical Institute, Slovak Academy of Sciences, Slovak Republic http://gpi.savba.sk/
† Dept. of Informatics, University of Bergen, Norway http://www.ii.uib.no/~tors

Mostly in the context of message passing on distributed memory parallel systems (see [2, 3, 9] etc). In [1] Feil et.al. discuss different parallelization strategies for message passing programming as well as shared memory programming. This paper builds on these works and extend it by incorporating the entire compression process and implementing it on an Origin 2000 system which has physical distributed memory, but allows the shared memory programming model as it provides cache coherent across the entire global memory.

This paper is organized as follows. In section 2 we describe the compression/uncompression algorithm as it is applied in the context of FD-scheme for the ground motion equations. We also provide some detailed timing of the separate parts of the compression code for different problems sizes in order to pinpoint the computational bottlenecks.

In section 3 we describe our parallelization. While we in section 4 are presenting and discussing the results from our computational experiments. In section 5 we wrap it all up with conclusion and directions of future work.

2. The compression algorithm. The data to be compressed are 2d arrays of 8-byte floating point numbers. The underlying idea of the compression algorithm is to first transform the data into wavelet-space using a 2d-FWT and then storing only the non-zero wavelet coefficients [8]. To achieve a substantial compression rate two techniques are used.

Thresholding: What we have is approximate values to inaccurate data. Thus all data less than a certain value should be regarded as noise and could be represented by zero without loss of significance. The noise level does of course depend on your problem, the granularity of your discretization and the order of this approximation. For the earthquake model we have found that setting equal to zero all coefficients less than $2^{-N}\texttt{Umax}$, where `Umax` being the largest wavelet coefficient, does not give any measurable difference in the overall solution for $N \geq 14$.

Quantization: Thresholding implies that we introduce an additional absolute error less than $2^{-N}\texttt{Umax}$ on the smallest coefficients. If we are willing to except the same absolute error on the reminding non-zero elements we can store only the N first digits of the mantissa and taking the reminding ones to be zero. Thus the wavelet coefficients can be represented by only N bits, giving us an extra saving factor of N/64. This is implemented by converting the floating point numbers into integer and than in the encoding phase only storing the last N bits of the non-zero integers.

Encoding/Decoding: After the wavelet coefficients have been massaged by thresholding and quantization they are encoded. This is done a by traversing the array of wavelet coefficient in linear order and pack the N bits of the non-zeroes into an integer vector. The position of a non-zero coefficient is relative to the previous non-zero and stored in the same vector as the coefficient itself. This makes the encoding a truly sequential process. You can't encode coefficient number $i+1$ before the value and position of coefficient number i is in place. Similarly for the decoding, you can't unpack $i+1$ before i is unpacked and you know its position. To minimize the use of space, we store the positions using only $m = \lceil \log_2 gap \rceil$ bits, where gap is the maximum distance between 2 non-zeroes. The programming of the encoding/decoding process relies heavily on the bitmap manipulation functions of Fortran90.

The 2d-wavelet transform: Our wavelet routine only works for arrays $m \times n$ where m and n are integer power of 2. This is not an inherit feature of the FWT. In general this constrain could be relaxed to m and n being divisible by 2^k where k is the decomposition depth. In our implementation we therefor first chop up the array

TABLE 1
Timings in sec. for the sequential compression algorithm.

	256x256	384x384	512x512	768x768	1024x1024	1536x1536	2048x2048
Compress	0.043	0.092	0.153	0.341	0.699	1.493	3.225
Wavelet	(55%)	(61%)	(59%)	(63%)	(67%)	(65%)	(71%)
Uncompress	0.033	0.065	0.108	0.236	0.527	1.109	2.581
Wavelet	(79%)	(73%)	(76%)	(77%)	(80%)	(77%)	(82%)

in `Nfrag` blocks of (different) power of 2 sizes [1]

The overall compression algorithm is displayed in Figure 1.

FIG. 1. *Algorithm: compression*

```
input:  UU, Nfrag, N
/* Do the wavelet transform on all blocks of UU */
for i = 1,Nfrag
    wavelet2d(block no.  i of UU);
endfor
Umax = $\max_{\forall(i,j)} | UU(i,j) |$;
where($| UU | < Umax \times 2^{-N}$) UU = 0 ;/* Thresholding */
UUI = UU /* quantization; */
encode (UUI);
```

When uncompressing the data again, we reverse this process, but now thresholding and quantization is no longer needed. The inverse wavelet transform is computationally essentially the same as the forward one.

In Table 1 we show some timings of the sequential algorithm. These timings document two important features of the algorithm

(i) It is linear in the terms of the problem size.

(ii) The computationally most expensive part is the wavelet transform which account for 60-80 % of the total time. The percentage being smallest for the compression as the reminding work are more in this case.

The significance of (i) is that the algorithm is data intensive and consequently efficient memory management is crucial to good performance on todays hierarchical memory system supercomputers. (ii) shows that any successful parallelization of this compression algorithm has to incorporate an efficiently parallelization of the 2d-wavelet transform.

3. Parallelization.

Our parallel programming model is that of Shared Memory Programming(SMP). Under this model we assume that all data is available to all processors. When looking for opportunities for parallelization all we have to look for is independent operations. That is; no processor writing into the same memory location simultaneously.

3.1. The 2d-wavelet transform.

The wavelet transform possesses 2 level of parallelization. An outer level over each piece of the array. This correspond to straightforward parallelization of the the **for**-loop in the algorithm displayed in Fig 1. Unfortunately this parallelism suffer from limited scalability and poor load balance,

[1]This correspond to writing m as with binary digits as $m = b_s b_{s-1} ... b_0$, $b_i \in \{0,1\}$ since this is equivalent to $m = \sum_{i=0}^{s} b_i 2^i$. All non-zero b_i then contribute to one block in the actual direction. The total number of 2d-blocks being the product of nonzero binary digits in m and n.

FIG. 2. *Algorithm: The 2d Wavelet algorithm*

```
input:  UU /* UU M x N matrix */
for j = 1,N
    wavelet1d(UU(:,j)); /* Multiple Column transform */
endfor
for i = 1,M
    wavelet1d(UU(i,:)); /* Multiple Row transform */
endfor
```

as the number of pieces, `Nfrag` is rather small and the pieces are of unequal size. We have therefore chosen to seek for parallelism within the wavelet routine itself and run the **for**-loop sequential. This is easy, provided we have a routine for the 1-d wavelet transform. The 2d-FWT might than be describe as in Fig 2.

As is evident from the above description each iteration of the for-loop operates on different data, thus there is no dependencies between the individual iteration, and they might be executed in parallel. It is however necessary to synchronize between the two loops, to make sure the different row transforms operates on data already massaged by the column transform.

Moreover, if memory is physically distributed then data, perfectly distributed for the column transforms, has the worst possible distribution for the multiple row transforms. The datamotion enforced by the row transforms correspond in essence to a matrix transpose. Each row is read and moved into a column vector. This could be done explicitly, by putting in and explicit call to the transpose of `UU`, and than just repeated the first **for**-loop. We've found this not to be beneficial on our system.

3.2. Thresholding, quantization and encoding/decoding. Thresholding, quantization and encoding require 3 pass through the data. In addition an initial pass to find the maximum of the absolute value of the wavelet coefficients is required. To prepare for the encoding we also need to find the maximum number of consecutive zeroes (after thresholding). This can be included in the pass for thresholding or quantization. It does however make the pass with the zero-finding-sequence into a very sequential operation.

There is two operations here that is not trivially parallel. It is the search for the longest sequence of zeroes and the encoding itself. These are inherently sequential. Our parallelization strategy is therefore to split the array in `Nproc` equal pieces (`Nproc` being the number of available processors) and run the compression routine independently for each part. This does not alter the accuracy (or loss thereof) of the wavelet coefficients, but it might change the compression rate as this depends on the length of the number of consecutive zero-elements. For our test data we have found a moderate increase in the the compression rate. This does not have any influence on the running time.

3.3. The data access pattern. The algorithm do requires two different data access pattern to the 2-d array which are to be compressed. The wavelet transform splits the data into 'power-of-2' blocks and access these blocks sequentially. The parallelization is applied to each of these blocks, thus for parallel execution each of these blocks is accessed columnwise by different processors. While in the encoding phase, the entire array is treated as one unit which also is accessed column wise.

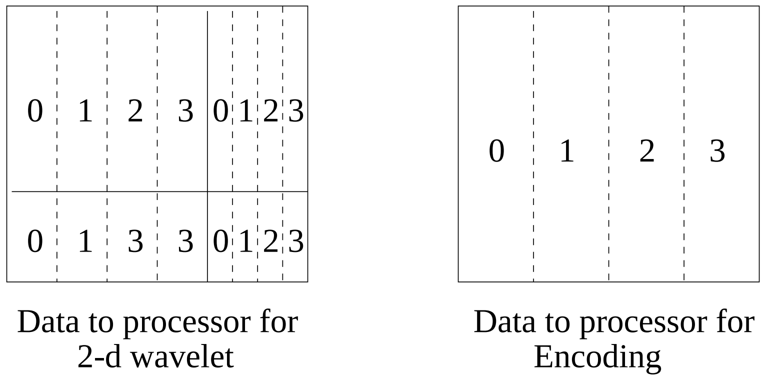
0 1 2 3 0 1 2 3
0 1 3 3 0 1 2 3
Data to processor for 2-d wavelet
0 1 2 3
Data to processor for Encoding

FIG. 3. *The data access pattern for 4 CPUs on a 384x384 arrays. The 2-d wavelet transform to the left and the encoding to the right*

For a UMA (Uniform Memory Access) shared memory system this should (in principal) have no effect on the computation speed. On a NUMA (Non Uniform Memory Access) it will. Because distributing data for shortest possible access path in one part of the algorithm inevitably force non-local access in the next part. An example of this is displayed in Figure 3.

Our testbed is an Origin 2000 system. On this system the operating system takes care of the data layout. It uses a "first touch" policy. In our case we have generated the test data the same way as we access it for the encoding and consequently we expect the data placement to be near optimal for the encoding phase.

4. Implementation and experiments. In our implementation we have used the newly created industry standard, OpenMP, [7] [6] to implement our SMP parallelization. This works by inserting compiler directives into the code in order to tell the compiler where it safely can run multiple task concurrently. There are two different kinds of parallel region. (i) Loops with independent iterations and (ii) Independent code blocks. In our case we have only used loop-parallelization.

The code has been implemented in Fortran90, and all experiments have been conducted on an Origin 2000 with 195 Mhz MIPS 10000 [10]. For the 2d-FWT we have used a depth of 4 and a filter type/length of 12.

Running time for the sequential code is presented in section 2, so here we'll only present speed-up numbers. In Figure 4 we present running time for 3 different problem sizes. The timing is splitted in compressing and uncompressing. As usual we see that the speed-up tends to tail off as the number of processors increase, and that the largest problem has the best speed-up. The speed-up compares very favorable with the numbers presented in [1]. As we have had no access to their code, we can not tell whether this is due to differences in computer system, choice of wavelet parameter or details in algorithm or implementation.

The next two figures show details of two different problem size. For the 2048x2048 case the different parts of the algorithm shows similar speed-up, while in the 1792x1792

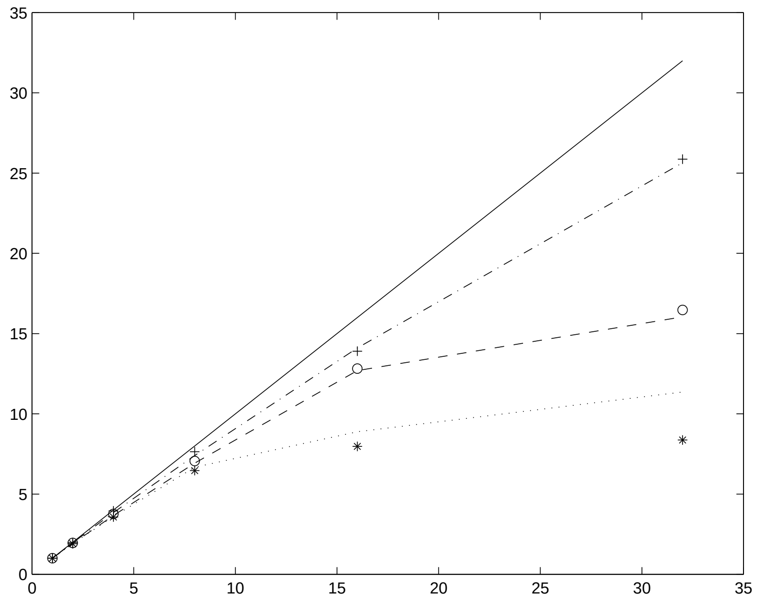

FIG. 4. *Speed-up curves for 3 different problem size. For compressing dashed line is 1024x1024, dotted line is 1792x1792 while dash-dot shows 2048x2048. The uncompressing is shown by circles for 1024x1024, stars for 1792x1792 and pluses for 2048x2048.*

case the wavelet-transform tends to hit the ceiling at a much earlier stage than the reminding part of the code. The reason for this is that in this case the wavelet-transform is performed on 9 blocks of the array, the largest one being 1024x1024 and the smallest one only 256x256. Thus only for the largest one of these we can expect speed-up similar to the 1024x1024 case. For the reminding blocks, which account for almost 2/3 of the data, we must expect the parallel efficiency to be less. This explains why the 1792x1792 case shows poorer overall speed-up than the smaller 1024x1024 case.

For the decoding we observe superlinear speed-up. The decoding, as well as the encoding, needs slightly above $8M \times N$ bytes of storage. On our system each CPU has 4 MB of level 2 cache, thus as we increase the number of CPUs more of the data fits into cache, and eventually all of data needed fits into cache. In our tests the decoding comes immidiately after the endoding. As the decoding access the same data as the encoding it benefites greatly from this order of computation as the cost of moving data from main memory to cache is paid by the encoding routine. As this part of the code needs no synchronization and is perfectly loadbalanced, we therefor observe superlinear speed-up. Of course the other parts of the code also benefit from increased total cache size as more processors are added, but this is countered by slow-down factors as synchronization and none local memory access.

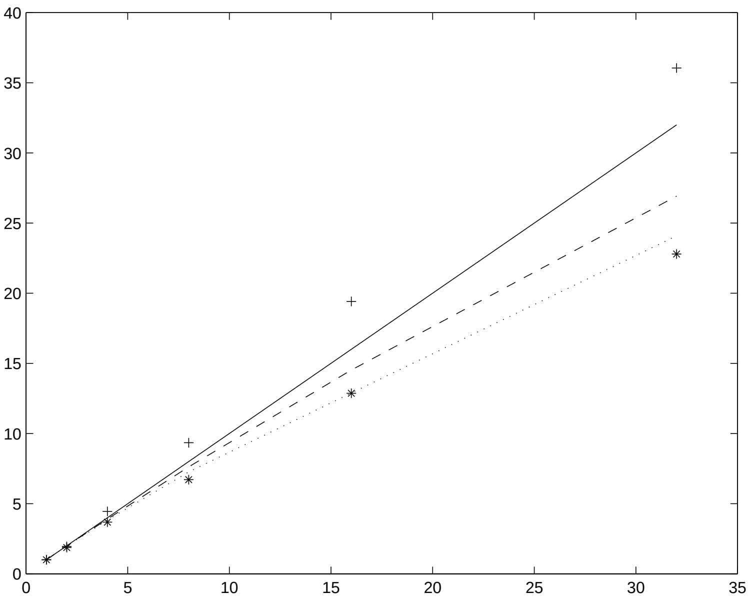

FIG. 5. *Speed-up curves for the different parts of the 2048x2048 case. Lines are for the wavelet-transforms, dashed for forward and dotted for backwards. The stars and the pluses are for the remaining components of the compression/uncompression respectively.*

In the 1792x1792 case the data is splitted into 9 different blocks of unequal sizes for the wavelet transform. Here we do expect less speed-up when compared to the two other cases. There is two reasons for this. As explained above the wavelet transforms operates on smaller arrays, making the parallelism more fine grained, and as the differences between the 1024x1024 and 2048x2048 cases shows, the larger the problem the better the speed-up on the 2d-wavelet transform. The second problem is that in this case we encounter the problem of differences in data access between the FWT and the core encoding process, discussed in section 3.3. Having a cc-Numa architecture, this could potential produce severe, slow downs at the Origin 2000. The initial data layout does in our case confirm with the optimal layout for the encoding process, and as the detailed timings shows the speed-up for this part is as good as for the two other cases. The 2d-wavelet transform does, as expected, not show the same excellent speed-up in this case, but by and large we find the Origin's behavior on the 1792x1792 case to proof that this computer do the remote memory access quite efficiently.

5. Conclusions.

In this paper we have investigated the parallelization of wavelet based compression routines. We found it is quite easy to produce a parallel SMP-style code using OpenMP. This program style hides the need for explicit coding of the communication, when accessing remote data, but it does not remove remote data access. This could potential create severe bottleneck on a distributed shared memory system like the Origin 2000. We're happy to say this did not seem to the case for our problem.

To achieve high performance we found sequential cache optimization also to be of great importance. Since cache optimization gives good datalocality we found that it also tends to increase the parallel speed-up as it reduced the need for non-local memory access.

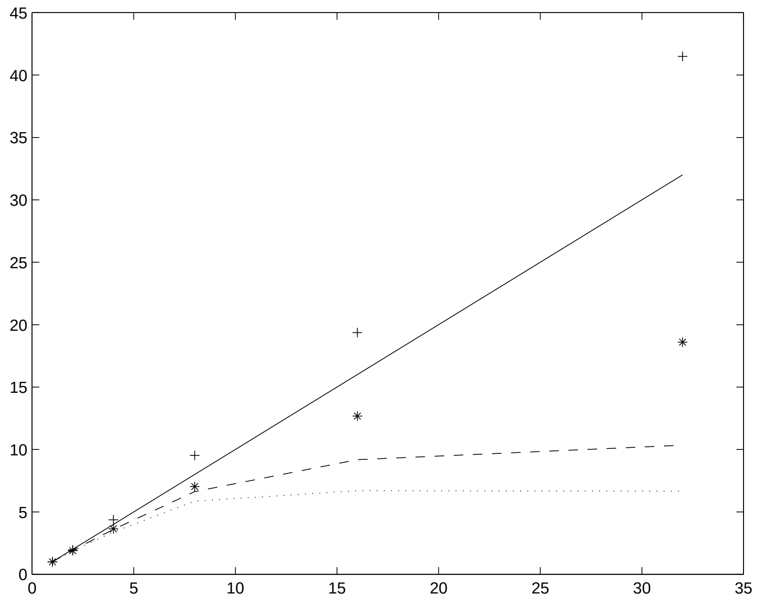

FIG. 6. *Speed-up curves for the different parts of the 1792x1792 case. Lines are for the wavelet-transforms, dashed for forward and dotted for backwards. The stars and the pluses are for the remaining components of the compression/uncompression respectively.*

The most time consuming part of this algorithm, the 2d-wavelet transform, has as discussed above, 2 levels of parallelism. This is not exploited in our implementation. We are currently investigating ways of doing this.

We are also working on integrating the compression routine in the full earthquake model and parallelizing this.

Acknowledgment. It is with great pleasure we acknowledge Jozef Kristek's contribution to this work. He wrote the first sequential version of the compression code. He has also contributed by stimulated and clarifying discussions on how the code works.

We are also happy to acknowledge valuable comments from an anonymous referee, which has greatly improved the quality of this paper.

This project has been supported by a grant from the Norwegian Supercomputing Program for computing time on the Origin 2000 and by a Collaborative research grant from NATO.

REFERENCES

[1] Feil M., Kutil R., Uhl A., Parallel Wavelet Transform on Multiprocessors Europar 99, Lecture notes in Computer Science 1685, Springer Verlag., 1999, 1013-1017

[2] Woo M-L., Parallel discrete wavelet transform on the Paragon MIMD machine. Proceedings of the 7th SIAM conference on parallel processing for scientific computing, 1995, 3-8.

[3] Uhl A., Wavelet packet best basis selection on moderate parallel MIMD architecture, Parallel Computing 22(1), 1996, 149-158.

[4] Graves R.W., Simulating seismic wave propagation in 3D elastic media using staggered - grid finite differences. Bull. Seism. Soc. Am., 1996, (86) 1091–1106.

[5] Moczo P., Lucká M., Kristek J., Kristeková M., 3D displacement finite differences and a combined memory optimization. Bull. Seism. Soc. Am., 1999, 89, 69-79.

[6] OpenMP: A proposed industry standard API for shared memory programming. Technical report, http://www.openmp.org/, October 1997.

[7] OpenMP Fortran Application Program Interface, Ver.1.0. Technical report, http://www.openm.org/, October 1997.

[8] Watson A.B., Yang G.Y., Solomon J.A., Villasenor J., Visual Thresholds for wavelet quantization error, SPIE Proceedings, Vol.2657, Human Vision and Electronic Imaging, B.Rogowitz and J.Allebach, Ed., The Society for Imaging Science and Technology, 1996.

[9] Nielsen O.M., Hegland M., A Scalable Parallel 2D Wavelet Transform Algorithm, TR-CS-97-21, The Australian National University, December 1997.

[10] Technical Overview of the Origin Family, http://www.sgi.com/Products/hardware/servers/techonolgy/