# Many-core and PSPI: Mixing fine-grain and coarse-grain parallelism

*Ching-Bih Liaw and Robert G. Clapp*

## ABSTRACT

Many of today's computer architectures are supported for fine-grain, rather than coarse-grain, parallelism. Conversely, many seismic imaging algorithms have been implemented using coarse-grain parallelization techniques. Sun's Niagara2 uses several processing threads per computational core, therefore the amount of memory per thread makes a strict coarse-grain approach to problems impractical. A strictly fine-grain parallelism approach can be problematic in algorithms that require frequent synchronization. We use a combination of fine-gain and coarse-grain parallelism in implementing a downward continuation based migration algorithm on the Niagara2. We show the best performance can be achieved by mixing these two programing styles.

## INTRODUCTION

Seismic imaging problems lend themselves well to coarse-grain parallelism. Kirchoff migration can be parallelized by splitting the image space (and/or data space) over many processing units. Downward continuation based migration can be parallelized over frequency. Flavors of downward continuation and reverse time migration can be further parallelized over shot or plane wave. All of these parallelism methods can be described as 'coarse-gained'. Coarse-grained parallelism fits well the cluster computing of the last decade. Several exciting new architectures including Nvidia's Grahic's Precision Unit (GPU), IBM's cell, Field Programable Gate Arrays (FPGA), and Sun's Niagara platform are more aimed at a fine-grained parallelism model. These platforms can have threads in the 10s-100s often making coarse-grain parallelism impractical because of memory constraints. Early results (Pell et al., 2008) on these architecture's are promising but implementation can be challenging.

Downward-continuation based migration (Claerbout, 1995) is a more challenging imaging algorithm to implement on a fine-grained parallel machine. The challenge in the implementation comes from the 2-D (shot-profile, plane-wave) 3-D (common-azimuth), or 4-D (narrow-azimuth, full-azimuth) FFT. The implicit-transpose and the non-uniform data access pattern does not easily port to FPGA and GPU solutions. The multi-thread per core approach of the Sun Niagara2 offers an easier parallelism route.

In this paper we demonstrate that the optimal solution for PSPI migration on the Niagara2 is by mixing the coarse-grained and fine-grained parallelism models. We begin by presenting an overview of the Niagara2 architecture and the PSPI algorithm. We show how some portions of the PSPI algorithm benefit from Niagara's multiple threads per core while others show only minimal improvement. We conclude by discussing the bottlenecks to further efficiency improvements.

## NIAGARA2 OVERVIEW

Niagara2 is the second generation innovative CMT, Chip Multi-Threading, CPU design from Sun Microsystems, Inc. It has eight computation cores with 4 Megabytes of shared L2 cache and 4 dual channel FBDIMM memory controller. It also contains integrated networking units, PCI-Express unit, embedded wire-speed cryptography coprocessor, and built-in virtualization supports. Each core has two integer execution pipes and one floating point execution pipe shared by eight fine-grained hardware threads. In all, Niagara2 sports 64 hardware threads and combines all major server and network functions on a single chip and is well suited for power efficient secure data-center and thread level parallel computing applications.

The idea behind the chip design is that most applications are memory bound, most of the time is waiting to retrieve memory from the either cache or main memory. By having several (in this case eight) simultaneous tasks attached to each processing unit you can hide the memory latency. Figure 1 illustrates this concept. The 'M' shows a thread waiting for a memory request while the 'C' shows computation. At each clock cycle computation is being performed and the time associated with memory requests are hidden.

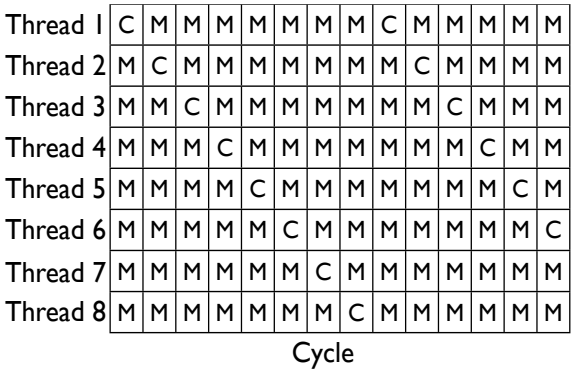


Figure 1: The idea behind the Niagara architecture. The 'M' shows a thread waiting for a memory request while the 'C' shows computation. At each clock cycle computation is being performed and the time associated with memory requests are hidden. [**NR**]

The Niagara2 platform performs well on an application when two requirements are met. First, that the problem is truly memory bound. This is a function of memory access speed, memory hierarchy, and the compute engines of a given core. Second, the parallelism granularity of the application cannot require a siginificant level of synchronization.

## PSPI MIGRATION

Downward continued migration comes in various flavors including Common Azimuth Migration (Biondi and Palacharla, 1996), shot profile migration, source-receiver migration, plane-wave or delayed shot migration, and narrow azimuth migrations. For downward continued based migration there are four potential computational bottlenecks that vary depending on the flavor of the downward continuation algorithm. The Phase-Shift Plus Interpolation (PSPI) method is one of the easier methods to implement. The computational cost is dominated by the cost of downward propagating a wavefield at a given frequency $w$, a given depth step $z$. Within this loop the wavefield is Fourier transformed, a correction term in the FX domain is applied, and the wavefield is downward continued in the FK domain. Pseudo code for the algorithm takes the following form,

```
Loop over w{  !CORASE
  Loop over z{
    Loop over source/receiver{
     Loop over v{
      FX    !FINE
      IFFT  !FINE
      FK    !FINE
    }
    FFT     !FINE
  }
  }
}
```

In many cases the dominant cost is the FFT step. The dimensionality of the FFT varies from 1-D (tilted plane-wave migration (Shan and Biondi, 2007)) to 4-D (narrow azimuth migration (Biondi, 2003)). The FFT cost is often dominant due its $nlog(n)$ cost ratio, $n$ being the number of points in the transform, and the non-cache friendly nature of multi-dimensional FFTs. The FK step, which involves evaluating a square root function and performing complex exponential is a second potential bottleneck. The high operational count per sample can eat up significant cycles. The FX step, which involves a complex exponential, or sine/cosine multiplication, has a similar, but computationally less demanding, profile.

## RESULTS

For this test we used a Common Azimuth Migration (CAM) variant of the PSPI algorithm discussed above. The FK, FX, and FFT are all performed on a 3-D field. We began from a code that used coarse-grain parallelization over frequency. We used a relatively small domain size (574x256x52) which is well beyond the L2 cache of the system but still allowed a large series of tests to be run in a reasonable amount of time.

Figure 2 shows the normalized performance of the entire algorithm as a function of coarse-grain threads. Note how we achieve linear speed up all the way to 9 threads. Going beyond 9 coarse threads was not possible given the machine's memory.

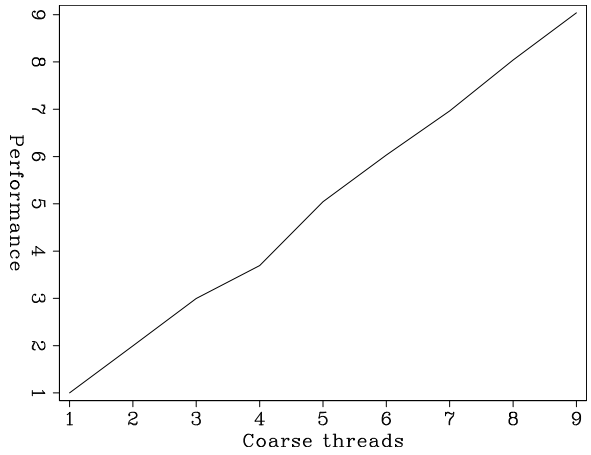


Figure 2: Performance as a function of the number of coarse-grained threads. [**NR**]

We then parallelized the `FX, FK`, and `FFT` routines. The `FK` and `FX` routines are sample by sample operations well suited to fine-grain parallelism and generally trivial to parallelize using the pthreads library. For the `FFT`, we used Sun's prime factor FFT rather than FFTW. The single-thread performance of the Sun's library was nearly double FFTW's performance. Figure 3 shows the normalized performance as the number of fine-grain threads increase. Note how we achieve nearly no performance gain after 32 threads. Figure 4 explains the lack of improvement. It shows the performance of the FFTW, FK, and FX steps portion of the algorithm. After 20-25 threads the FFT shows no performance improvements. This is not surprising due to the synchronization inherent in the FFT algorithm.

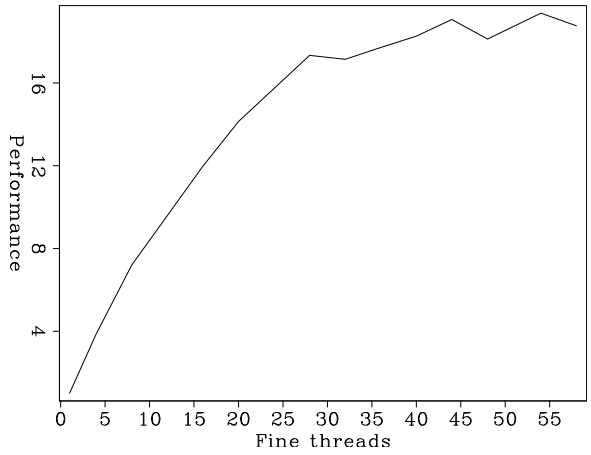


Figure 3: Normalized performance as a function of the number of fine-grained threads. Note that little performance gain is achieved after 32 fine-grain threads. [**NR**]

As a final test we combined the coarse-grained and fine-grained approaches. Figure 5 shows the maximum performance as a function of coarse-grain threads $n_c$. For each coarse-grain thread we used $n$ fine-grain threads where $n = floor(63/n_c)$, maximizing the available threads on the machine. Note that the graph is normalized

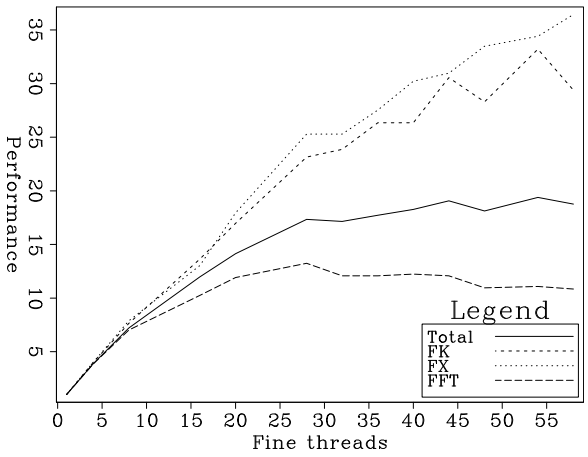


Figure 4: Performance as a function of the number of fine-grained threads for different parts of the algorithms. Not the nearly linear speed up of the FK and FX steps while performance peaks for the FFT at 20 threads. [**NR**]

by the single thread performance. Peak performance was achieved using 6 or more coarse-grain threads. Figure 6 shows the break down by function. Not surprisingly, the FK and FX step show nearly constant performance independent of the number coarse-grain vs. fine-grain threads. On the other hand the FFT benefits from less fine-grained parallelism bringing up the overall total performance of the algorithm.

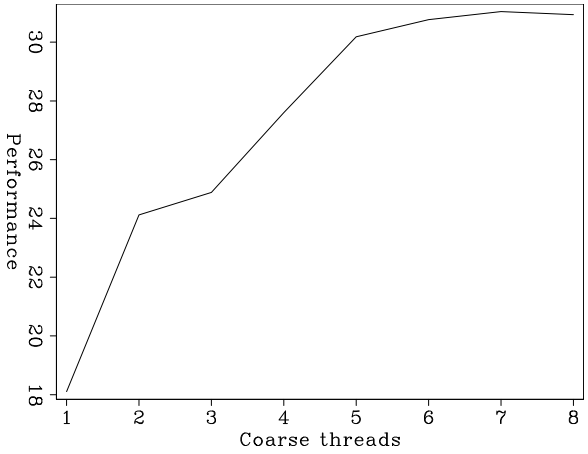


Figure 5: Performance as a function of number of coarse-grain threads. The number of fine-grain $n_f$ threads per coarse-grain $n_c$ threads is $n_f = floor(63/n_c)$. [**NR**]

As a comparison we run the same code on 4-core 1.8GHz, dual processor intel machine. Running 8 coarse-grain threads and normalizing in terms of the the Niagara2 single thread results we found:

| **Segment** | **Relative speed** |
|---|---|
| FFT | 23.9 |
| FX | 10.9 |
| FK | 61.8 |
| Total | 23.9 |

In general the code scaled linearly with number of processors.The `FX` ratio was low compared to the Niagara because of the significant memory requests (due to the

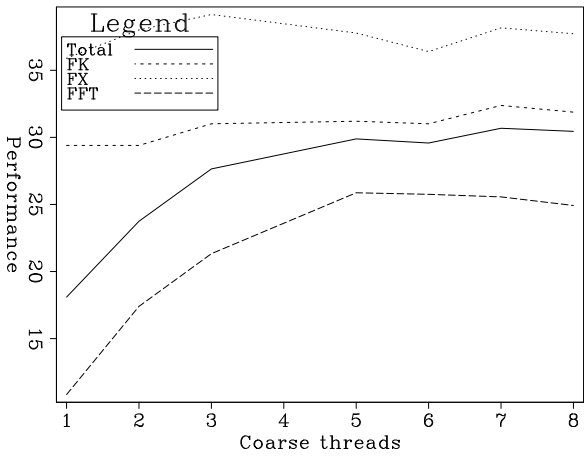


Figure 6: Performance of different routines with the PSPI algorithm as a function of the number of coarse-grain threads. Note how the FFT benefits the most from more coarse-grain parallelism. [**NR**]

velocity correction), something that the Niagara2 archetecture is well designed for. The `FK` number was large due to the vector nature of the computation and its high floating point operation count.

Improving the floating point and vector potential of the Niagara architecture could have a large impact on these results. Both the FFT and the FK steps involve significant floating point computations. The synchronization requirements of the FFT algorithm limits effective scaling to approximately 16 threads even for large volumes.

## CONCLUSIONS

We implemented PSPI migration on the Sun Niagara2 by combining coarse-grained and fine-grained prallelism. We showed that the multi-thread per core model leads to significant uplift in performance over a single thread approach. Compared to a strictly fine grain parallelism we achieved a 60% uplift. Compared to a coarse grain approach the improvement was 5X. Improved floating point/vector performance could lead to signficant uplift for this aglorithm.

## REFERENCES

Biondi, B., 2003, Narrow azimuth migration of marine streamer data: SEG, Expanded Abstracts, **22**, 897–900.

Biondi, B. and G. Palacharla, 1996, 3-d prestack migration of common-azimuth data: Geophysics, 1822–1832, Soc. of Expl. Geophys.

Claerbout, J. F., 1995, Basic Earth Imaging: Stanford Exploration Project.

Pell, O., T. Nemeth, J. Stefani, and R. Ergas, 2008, Design space analysis for the acoustic wave equation implementation on fpga circuits: European Association of Geoscientists and Engineers, Expanded Abstracts, 1406–1409.

Shan, G. and B. Biondi, 2007, Angle domain common image gathers for steep reflectors: 2007, **131**, 33–46.