Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems*

Oreste Villa[1], Daniel Chavarría-Miranda[1], Vidhya Gurumoorthi[2], Andrés Márquez[1], and Sriram Krishnamoorthy[1]

[1]High-Performance Computing
Pacific Northwest National Laboratory
{oreste.villa, daniel.chavarria, andres.marquez, sriram}@pnl.gov
[2]Applied Computer Science
Pacific Northwest National Laboratory
vidhya.gurumoorthi@pnl.gov

Abstract

Floating-point operations, as defined in the IEEE-754 standard, are not associative. The ordering of large numbers of operations (such as summations) that deal with operands of substantially different magnitudes can significantly affect the final result. On massively multi-threaded systems, the non-deterministic nature of how machine floating-point operations are interleaved, combined with the fact that intermediate values have to be rounded or truncated to fit in the available precision leads to non-deterministic numerical error propagation. We have investigated on a Cray XMT system the effect of non-deterministic error propagation by observing the convergence rate of a conjugate gradient calculation used as part of a Power State Estimation (PSE) application. As a possible mitigation strategy, we have explored quadruple precision accumulation, as well as a deterministic parallel tree scheme. The tree based approach has consistently outperformed the quadruple precision approach due to an improved convergence rate. As a consequence, we motivate the need for compile time mechanisms that enable enforcement of parallel deterministic operations on the Cray XMT.

*This work was funded by the Center for Adaptive Supercomputing Software - MultiThreaded Architectures (CASS-MT) at the U.S. Department of Energy's Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

1 Introduction

Most scientific applications use floating point arithmetic for their numerical calculations. Floating point arithmetic is known to be non-associative since the limited precision of the representation requires intermediate values be rounded. The IEEE-754 [1] standard is the de facto industry standard floating point representation used by almost all applications and platforms. This standard provides uniform semantics for operations across a wide range of implementations. The standard defines correct behavior for all operations, as well as any necessary rounding. IEEE floating point numbers have a finite precision mantissa and a finite range exponent, a property that requires rounding in the intermediate stages of arithmetic calculations, as for instance during long accumulations. This limited precision representation makes floating point arithmetic non associative. While techniques exist for maintaining acceptable precision in the middle of long sequences of floating point calculations [10], [12], they almost always require algorithmic changes to the given computation. As a result, the limited-precision and non associativity of IEEE floating point is generally accepted as a reasonable tradeoff to permit fast hardware implementation of floating-point arithmetic. More generally when rounding errors need to be minimized, one of the "brute force" approaches is to modify the precision at which the application is executed, for instance changing it precision from single to double and from double to quadruple. This approach is an attempt to reduce the magnitude of the problem's manifestation, rather than fully eliminate it. As a consequence, portable floating point computations must always be performed strictly in the order specified by the sequential evaluation. This makes impossible to parallelize most floating point operations without strictly respecting the IEEE standard.

This problem is particularly evident in systems when many different concurrent threads accumulate partial sums using shared variables. In systems where the thread "ordering" changes dynamically (at system level), for different executions of the same accumulation, different results will be produced. In calculations such as those in iterative solvers, as the results are propagated trough various iterations, the final result can differ in significant digits accross different executions. This is specially true if the problem is numerically ill-conditioned. Convergence problems can arise and total execution time can change due to a potential increase in the number of iterations. Debug procedures based on the validation of intermediate results became impossible as the results change over multiple runs.

Several applications rely on the Conjugate Gradient (CG) method which is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite. The conjugate gradient method is an iterative method, so it can be applied to sparse systems which are too large to be handled by direct methods such as Cholesky decomposition. These systems arise regularly when numerically solving partial differential equations such as in power grid analysis applications i.e. Power State Estimation (PSE). Sparse Matrix-Vector Multiply (SpMV) is the dominant computation kernel in CG. In SpMV a dot product is computed between the rows of a matix A and a vector x which effectively requires to sum the products of the nonzero matrix values with their corresponding vector entries in x. For sparse matrices, the number of non-zero entries per row can be unbalanced, with some rows requiring sums of only a few products, and exceptional rows requiring much larger sums. In addition to the SpMV calculation, a typical CG iteration requires a few global dot products with length equal to the size of the vectors. For large numerical calculations, if these summations must be serialized, they can become a major performance bottleneck in the task, limiting the benefits of parallelism.

This paper investigates the effect of the problem of non deterministic accumulation on the convergence of a conjugate gradient calculation used as part of a power grid analysis application, i.e. Power State Estimation (PSE) on the Cray XMT system. We have investigated the common and not always deterministic solution of increasing precision of the operands form double to quadruple precision, as well as a custom library-based tree scheme that performs accumulations deterministically maintaining the same precision. In the analyzed power grid application for the selected platform, the tree based approach performed faster and with absolute determinism respect to the quadruple precision approach.

The paper is organized as follows: Section 2 presents background material on the details of the floating point accumulation problem, the PSE application and the Cray XMT system. Section 3 presents the design of the PSE application on the Cray XMT. Section 4 discusses our experimental results, presents the deterministic mutlithreaded accumulation tree scheme and compares it to different levels of precision (namely double and quadruple) in terms of accuracy, determinism, and performance. Finally, Section 5 presents our conclusions.

2 Background

2.1 Non-Associativity of Floating-Point Accumulation

The non-associativity of floating point accumulation is originated by the limited precision and range of the IEEE floating-point representation. Figure 1 shows an example where the associativity problem arises. The example considers an hypothetical numerical representation with a precision of 32 "digits", showing two very large numbers added to a very small number – the large numbers are conceived to be at the extremes of the representation range. This example is only hypothetical and it does not reflect the true details of the IEEE representation. Under the assumption that the operands are added according to the left order in Figure 1, the first two numbers cancel each other out, reducing the dynamic range, such that the small number (0.01) is added to zero with the result "0.01". If the operands are added following the reverse order (Figure 1 on the right), the addition of the large negative number to the small number gives as a result the large negative number itself because the dynamic range is not reduced by the first operation. The final summation

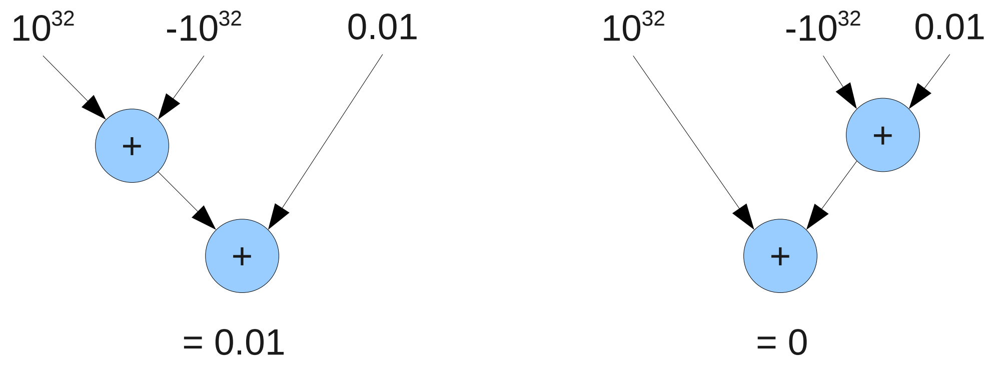

Figure 1: Non-Associativity of Floating-Point Accumulation.

result is hence "0". As the example shows, the two different associations yield different results. Under the assumption that the above accumulation is performed in parallel using three concurrent threads the result is non-determinate due to non-deterministic error propagation. Intuitively, as the amount of operations and threads grows, the number of thread interleave permutations grows with the consequence of multiple result outcomes.

2.2 Power State Estimation

The static Power State Estimation (PSE) problem [2] can be defined as follows: given power grid topological information, under-determined telemetered line power flows as well as parameter data, compute an estimate of the system state, represented by bus voltages. The most commonly used method for computing state estimation is the Weighted Least-Squares (WLS) method [11]. The WLS method tries to minimize the sum of the weighted squares of the components of the residual vector r, $r = z - h(x)$, where x is the approximated system state, z is a vector of measured quantities and h is a vector function. WLS can then be expressed as:

$$\text{minimize } J = \sum_{i=1}^{m} w_i r_i^2 = r^T W_r \tag{1}$$

where w_i is the weight for the residual r_i, W is a diagonal matrix, m is the number of measurements.

In general, this is a non-linear problem, which is solved by the Newton-Raphson iterative procedure; every iteration requires solving a large set of sparse linear equations. The matrix form of the problem will be sparse, since it is derived from the topological information for the power grid: it depends on the number of buses (power grid nodes) and their interconnections. The number of nonzero elements in each row of the matrix varies greatly - unlike sparse matrices derived in the solutions of partial differential equations - and is badly conditioned [11]. The problem can be solved using sparse direct solvers, such as those based on LU factorization, or using iterative solvers such as those based on Conjugate Gradient (CG).

PSE is a critical element of the software used by grid control centers. To be useful for effective decision support by the grid operators, given the dynamic nature of grid and the constantly updated telemetry data, the PSE problem needs to be solved in less than ten seconds. For large grid configurations this requirement presents a challenge to the current commercial PSE codes which currently are unable to exploit parallel processor technology effectively.

2.3 Cray XMT

The Cray XMT is the commercial name for the shared-memory multithreaded machine developed by Cray under the code name "Eldorado" [7, 6]. The system is composed of dual-socket Opteron AMD service nodes and custom-designed multithreaded compute nodes with *Threadstorm* processors. The entire system is connected using the Cray Seastar-2.2 high speed interconnect. Figure 2 gives a high level view of the system. The *Threadstorm* processors are represented as gray boxes (running MTX, a BSD variant), while the colored boxes represent the AMD Opteron service nodes (running a standard

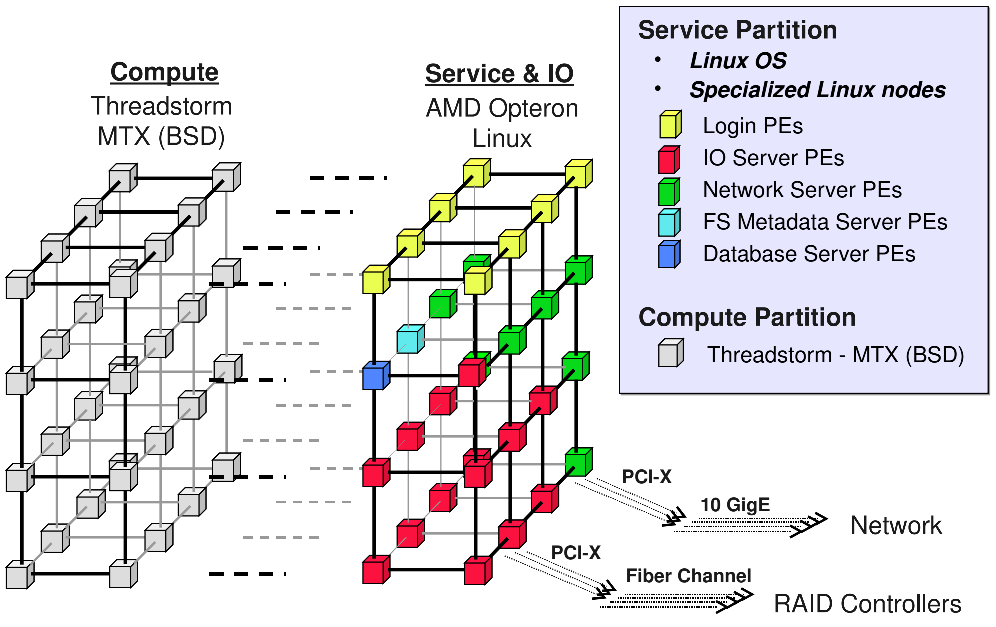

Figure 2: Cray XMT overall system architecture.

Linux OS). The XMT system can scale up to 8,192 *Threadstorm* processors and 128 TB of shared memory. Each Threadstorm processor is able to schedule 128 fine-grained hardware threads to avoid memory-access generated pipeline stalls on a cycle-by-cycle basis. At runtime, a software thread is mapped to a hardware stream comprised of a program counter, a status word, a target register and 32 general purpose registers. Each Threadstorm processor has a VLIW (Very Long Instruction Word) pipeline containing operations for the Memory functional unit, the Arithmetic unit and the Control unit[1].

Each Threadstorm is associated with a memory system that can accommodate up to 8GB of 128-bit wide DDR memory. Each memory controller is complemented with a 128KB, 4-way associative data buffer to reduce access latencies (this is the only data buffer present in the entire memory hierarchy). Memory is structured with full-empty-, pointer forwarding- and trap- bits to support fine grained thread synchronization with little overhead. The memory is hashed at a granularity of 64 bytes (see Figure 4) and fully accessible through load/store operations to any Threadstorm processor connected to the Seastar-2.2 network, which is configured in a 3D toroidal topology. While memory is completely shared among Threadstorm processors, it is decoupled from the main memory in the AMD Opteron service nodes. Communication between Threadstorm nodes and Opteron nodes is performed through a Lightweight Communication Library (LUC). On the Opteron side, LUC is implemented using Portals [5] whereas on the Threadstorm side it uses a Fast I/O API that is layered over the Seastar-2 native protocol. Continuous random accesses to memory by the Threadstorm processor will top memory bandwidth at around 100M requests per second. Up to 500M memory requests per second are delivered by the associated data buffers. (Figure 3).

The Seastar-2 is a full system-on-chip design that integrates six high speed serial links, a 3-D router, with network interface functionality. The network interconnect includes an embedded PowerPC processor, in a single chip. In the Seastar-2 network interface, there are two DMA engines, one for sending and the other for receiving, that interact with a router that supports a 3-D torus interconnect and the HyperTransport (HT) cave that provides an interface to the Cray Threadstorm processor and the memory. The embedded processor is

[1]The Arithmetic unit is capable of performing a floating-point multiply-add per cycle. In conjunction with the control unit doubling as arithmetic unit, a Threadstorm is capable of achieving 1.5 GFlops at a clock rate of 500MHz. A 64KB, 4-way associative instruction cache helps in exploiting code locality.

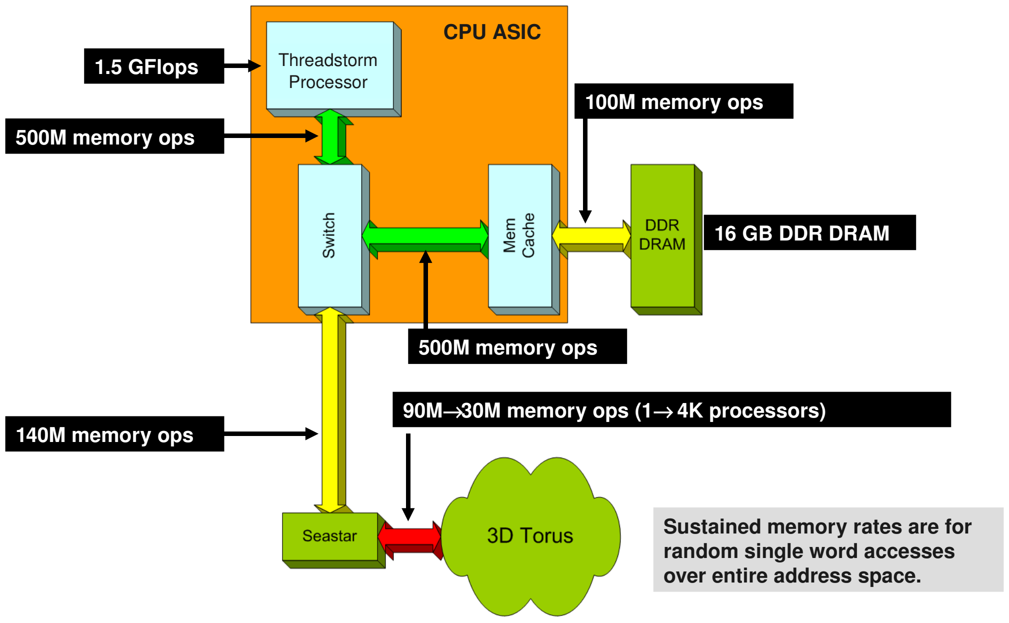

Figure 3: Threadstorm node with theoretical speeds.

provided to program the DMA engines and assist with other network-level processing needs, in particular supporting the Portals message-passing layer and load/store operations.

The software environment on the Cray XMT includes a custom, multithreaded operating system for the Threadstorm compute nodes (MTX), a parallelizing C/C++ cross-compiler targeting Threadstorm, a standard Linux 64-bit environment executing on the service and I/O nodes, as well as the necessary libraries to provide communication and interaction between the two parts of the XMT system. The parallelizing C/C++ compiler generates multithreaded code that is mapped to the threaded capabilities of the processors automatically. Parallelism discovery happens fully or semi- automatically by the addition of `pragmas` (directives) to the C/C++ source code.

2.3.1 XMT Compiler & Programming Model

The Cray XMT C/C++ programming environment focuses on detecting and taking advantage of loop-based data parallelism. Loop iterations are then mapped using different scheduling policies to threads on the processors. Data partitioning and access are not a problem at all, given the XMT's support for global shared memory access.

The basic execution model for applications on the XMT is based on fork-join parallelism. The execution of a program starts on a single thread and continues in the same manner until the first *parallel region* is encountered. At that point, the application will ask the runtime system to spawn some number of threads (depending on the resource requirements of the parallel region and also on how many processors is the application running on). Multiple parallel loops may reside inside the same parallel region, the compiler will have introduced synchronization points between parallel loops to preserve data dependencies.

This basic model of execution assumes that iterations of loops that have been parallelized are independent of each other, or that proper synchronization between them has been introduced by the programmer. There are certain cases in which dependencies that could prevent parallel loop execution are automatically handled by the compiler and runtime system. The two main cases are *linear recurrences* and *reductions*. *Linear recurrences* appear when the value to be computed in a particular iteration depends on values computed in previous iteration, in such a way that the dependence are simple and only cross a small number of constant iterations: e.g.

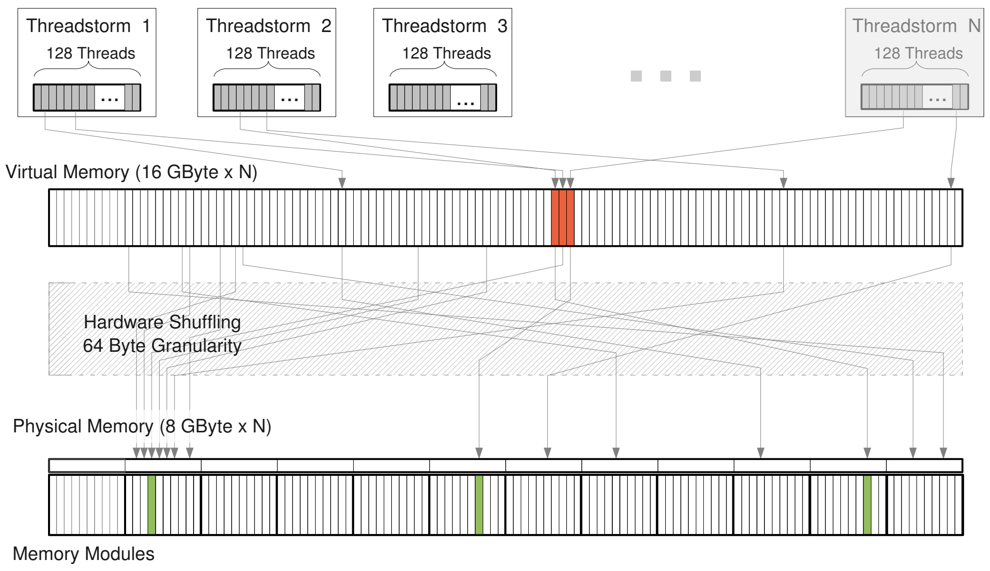

Figure 4: Cray XMT Threadstorm memory subsystem.

```
  for (i = 0; i < n; i++)
    x[i] = x[i - 1] + m;
```

This paper focuses on *reductions*. Reductions appear when iterations in a loop try to write their results to a single shared value. A typical example is a *sum reduction*:

```
s = 0.0;
for (i = 0; i < n; i++)
  s += x[i];
```

It is important to note that in most cases the number of iterations in a parallel loop (`n`) will be (much) larger than the number of threads executing the loop. If this is the case, then each thread can compute its portion of the reduction sequentially into a private location (partial sums in the example), and then these partial results can be combined to produce the final result (global final sum in the example). The following pseudo-code illustrates how this can be done:

```
s = 0.0;

for (tid = 0; tid < num_threads; tid++) {
  ls = 0.0;

  for (<iterations from i loop assigned
        to thread tid>)
    ls += x[i];

  <combine ls with global s>
}
```

The XMT compiler automatically generates the parallel pseudo-code presented above for reduction loops written in the original form.

This paper focuses on the combination of local by-thread results onto global results for floating-point reductions.

3 Power State Estimation on the Cray XMT

We ported a Fortran-based sequential WLS power system state estimator to the Cray XMT. This code uses a conjugate gradient (CG) sparse solver at its core, offering much better scalability than direct solvers based on LU or Cholesky factorization [11]. The application executes the following steps:

1. read input files describing the topological characteristics of the power grid to be analyzed, as well as a set of telemetry measurements to be used to compute the static state of the grid;

2. from the topological grid data, construct sparse matrix forms for the gain matrix and the H matrix (linear representation of the h vector function);

3. using the Newton-Raphson iterative method, update the values of the gain and H matrices, compute the right-hand-side, and call the CG solver;

4. check for convergence of the Newton-Raphson iterations;

5. finally, once the solution converges (it may not!), a complete state estimation for the input grid is ready.

Many of the steps involved in the WLS computation were parallelized either automatically by the XMT compiler, or by introducing directives to guide the parallelization. The input step [**1.**] is sequential, but it is executed only once for a given grid configuration and therefore is not on the critical path. On the other hand, the remaining steps need to be executed whenever a new set of telemetry data becomes available. The initial steps [**2.**] that construct the sparse matrix structures for the gain and H matrices have been parallelized, but they do not belong to the critical path of the program (95% of the computation time is spent in the Newton-Raphson WLS iterative solver). The critical steps inside the iterative loop [**3.** and **4.**], have been fully parallelized with special attention paid to the CG solver. The CG solver computes the solution to a linear equation $Ax = b$. x and b are dense vectors of unknowns and right-handside values, respectively. The matrix A is represented using the sparse compressed row form, where a 1D array is used to represent the non-zero elements of A; a row array indicates where each row begins and ends: i.e. $\texttt{row}_i$ indicates where the i-th row begins, $\texttt{row}_{i+1} - 1$ indicates where it ends; and a `col` array which indicates for a particular non-zero element what column does it belong to in its row. The matrix A is $N \times N$.

The solver then iteratively computes a solution to the system of equations. The main operation in the iterative computation is a sparse matrix-vector product. All the other steps are vector operations: addition/subtraction and dot products. The parallelization of the vector operations was straightforward. In all cases the compiler was able to parallelize the loops without the use of any XMT compiler directives. The sparse matrix-vector product required some directives in order to improve performance, but the compiler was able to recognize the loop nest as parallel without any guidance.

Figure 5 shows the C code used to compute a sparse matrix-vector product for the CG solver. `n` corresponds to the number of rows in the matrix, the array `a` contains the non-zero elements of the matrix, the array `irow` contains the entries in `a` where rows start and end, the array `icol` contains the column position for each element in `a`, and `x` and `r` correspond to dense vectors used as the factor and the result vector respectively. We used a compiler directive (`#pragma mta loop serial`) to indicate to the compiler that the inner loop iterating over the non-zero elements in a row should be run sequentially, while the outer loop should be parallel. Our preliminary experiments indicated that the performance of the code was better in this manner than with the alternative in which the compiler fused the two loops (`i` and `j`) into a single, flat parallel loop.

```
int i, j;
double t;

#pragma mta block dynamic schedule
#pragma mta use 100 streams
for (i = 0; i < n; i++) {
  t = 0.0;
#pragma mta loop serial
  for (j = irow[i]; j < irow[i + 1]; j++)
    t += a[j] * x[icol[j]];

  r[i] = t;
}
```

Figure 5: Sparse matrix-vector product

4 Experimental Results

In this Section we present our evaluation of the non-deterministic problem of parallel multithreaded CG calculation on the Cray XMT for the PSE application. We first show the magnitude of the problem in a double precision based code, we then investigate the solution of increasing the accumulators' precision form double to quadruple. We then present a custom tree-based parallel reduction scheme that performs reductions deterministically.

We tightened our PSE code to use statically scheduled parallel loops on the XMT using the compiler directive (`#pragma mta block schedule`), so that all the effects of dynamically scheduling loop iterations onto threads are suppressed. This guarantees deterministic assignment of iterations to threads. We performed this modification on each accumula-

WLS iteration	Run 1	Run 2	Run 3	Diff 2 vs. 1	Diff 3 vs. 1
1	1.64E+09	1.64E+09	1.64E+09	0.00%	0.00%
2	1.88E+09	1.88E+09	1.88E+09	0.00%	0.00%
3	3.29E+07	3.29E+07	3.29E+07	0.00%	0.00%
4	4.01E+05	4.01E+05	4.01E+05	0.02%	0.01%
5	1.50E+02	1.29E+02	1.24E+02	14.25%	17.63%
6	5.92E+00	5.13E+00	7.37E+00	13.30%	24.64%
7	5.22E-01	4.46E-01	4.59E-01	14.52%	12.06%

Table 1: Variability in double precision for the Euclidean norm at the end of each WLS iteration for different runs with the same number of threads and same input set.

WLS iteration	Run 1	Run 2	Run 3	Diff 2 vs. 1	Diff 3 vs. 1
1	1.64E+09	1.64E+09	1.64E+09	0.0000%	0.0000%
2	1.88E+09	1.88E+09	1.88E+09	0.0000%	0.0000%
3	3.29E+07	3.29E+07	3.29E+07	0.0000%	0.0000%
4	4.01E+05	4.01E+05	4.01E+05	0.0000%	0.0000%
5	1.43E+02	1.43E+02	1.43E+02	0.0000%	0.0000%
6	6.14E+00	6.14E+00	6.14E+00	0.0000%	0.0000%
7	5.73E-01	5.73E-01	5.73E-01	0.0000%	0.0001%

Table 2: Variability in quadruple precision for the Euclidean norm at the end of each WLS iteration for different runs with the same number of threads and same input set.

tion or reduction loop in the code. This unfortunately does not guarantee determinate results since the XMT compiler automatically recognizes the reductions and generates code that performs accumulations using atomically updated shared variables. In this case, the order in which different threads perform the atomic updates is influenced by run-time and OS thread scheduling policies. To quantitatively measure the phenomenon, we focused our attention on the computation of the Euclidean norm of the residual vector, used as convergence criterion for the CG loop. This a single highly observable scalar value that can be recorded before and after every iteration of the external WLS loop:

$$||v|| = \sqrt{\sum_{i=1}^{n} v_i^2} \qquad (2)$$

Table 1 shows the value of the Euclidean norm computed in double precision, for 3 different runs of the same code using the same number of processors, threads and input sets. The table presents the value of the Euclidean norm for different WLS iterations. The results were obtained on a 16 processor Cray XMT using 100 streams per processor for each parallel region. The PSE execution completed after seven iterations of the WLS loop. Table 1 shows the difference (in percentage) of the second and third runs with respect to the first one (used as an arbitrary baseline). We can see that different runs produce different results. We also notice that as the number of WLS iterations increases, the variability in the Euclidean norm increases in magnitude, although not proportionally and not uniformly. This is due to the propagation of different accumulation errors at different levels, from the linear solver to the external WLS loop.

A common technique used to reduce the effects of non-deterministic error propagation is to increase the precision of the representation. We modified our code to use quadruple-precision (128 bit) accumulators. This approach substantially reduced the variability of the norm value. Table 2 shows the same execution described above, using quadruple precision accumulation variables. The very small variability is limited to the last WLS iteration. We can say that, in general, quadruple precision does not solve the problem of non-deterministic error propagation but substantially reduces it. However, the tradeoff of quadruple-precision on a 64-bit architecture, is that it is has to be emulated (via combination of two double-precision variables), therefore the performance of the solution is compromised. As a reference, an accumulation of 28,000 elements (on

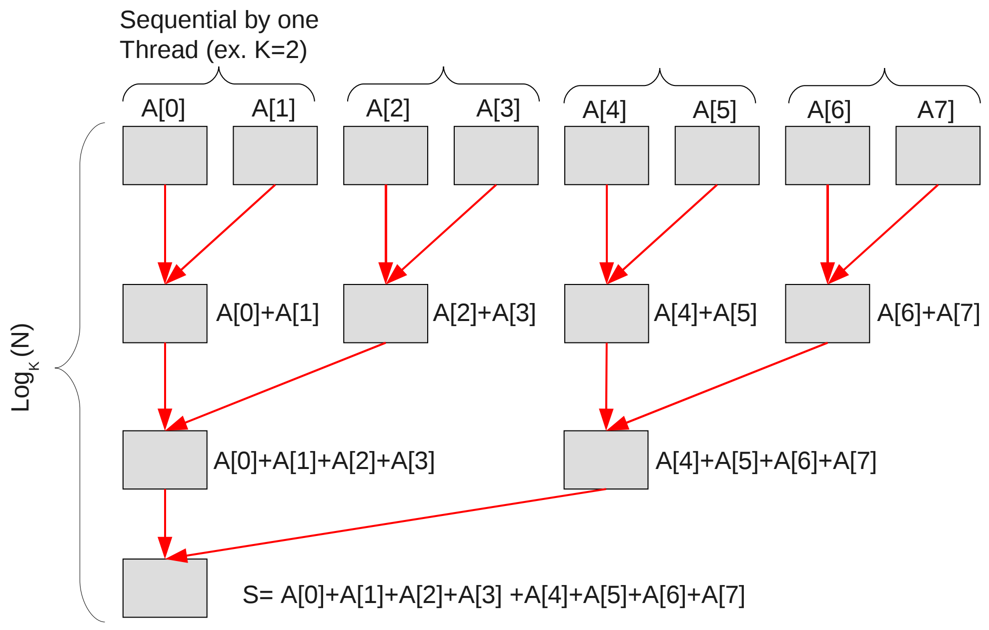

Figure 6: Tree reduction on the XMT.

16 XMT processors with 100 threads each) requires 0.519 milliseconds in double precision and 1.190 milliseconds in quadruple precision.

In general, provided that the iterative method converges, more than striving for an absolutely correct value (which in principle can be obtained only with infinite precision) it is more appealing to have determinate and repeatable executions. An approach to achieve this, is to utilize an alternative deterministic accumulation algorithm. We designed a software-based parallel-prefix accumulation that allows us to compute the result over N inputs in $O(\mathtt{log}_k(N))$ steps. This tree-based accumulation has been integrated in a library, and it is callable as follows:

```
acc = acc_tree(array, size_array, K);
```

Several styles of parallel prefix are found in practice (e.g. Brent-Kung [4], Sklansky [9], Ben-Asher [3], Nachiket [8]). The exact number of floating-point operations required depends on the style of prefix used, but is usually more than the simple, serial sum. Parallel-prefix allows to tradeoff extra computation for lower latency generation of results. Figure 6 shows a schematic of the reduction tree we implemented on the Cray XMT. The reduction is performed in different levels with a granularity equal to K. In each level, a single thread accumulates sequentially K different elements. Since the “shape” of the reduction tree depends only on K, for a fixed value of K, the reduction is deterministic but potentially less accurate than quadruple precision. The execution time is comparable to the default double precision accumulation performed by the compiler and our experiments show that it requires 0.635 milliseconds to accumulate 28,000 elements. Accuracy varies with degree K and not with processor and thread count, and it is comparable to the accuracy achieved by the compiler based double precision reduction.

An interesting property of the tree-based reduction is that as the degree K changes the performance and the accuracy of the solution varies. Figure 7 on the left shows the performance of the tree reduction for different values of K and for different processor counts. The number of requested threads for each processor is always 100. The accumulated input array is composed of 28,000 uniformly distributed random elements whit total sum of 2.69E18. We can see that there is a range for K (from 32 to 256) that guarantees a lower execution time. With larger values of K the number of sequential sums that each

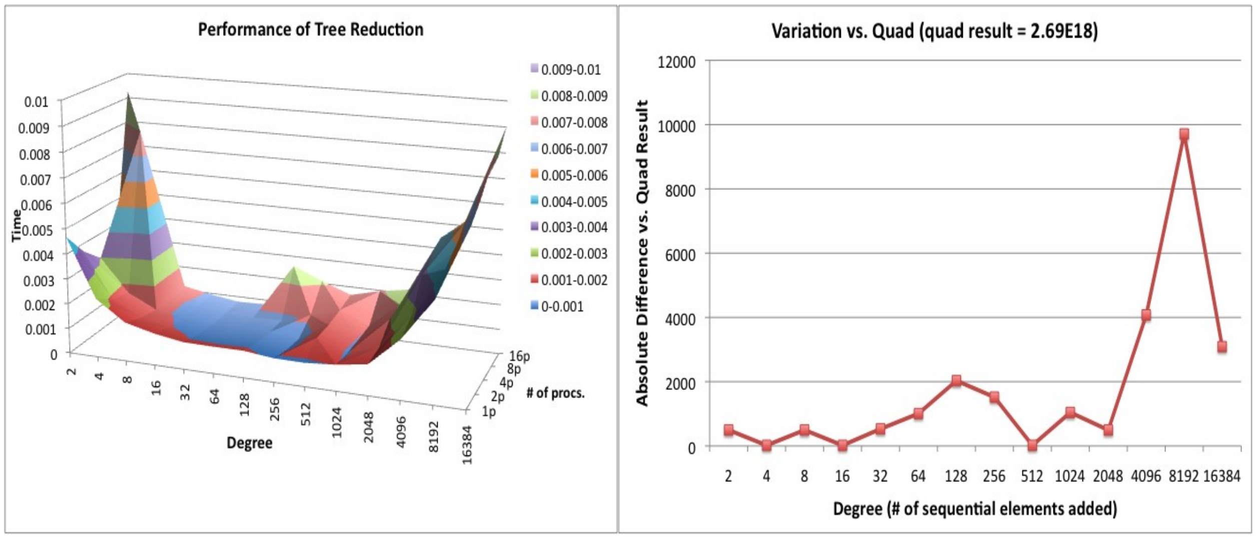

Figure 7: Performance and accuracy respect to quadruple precision of the tree solution for different values of K.

thread has to perform is too high so that the performance is limited by the sequential accumulations, while for small values of K the overhead of creating threads to accumulate just a few values, negatively impacts the performance.

Figure 7 on the right shows the relative accuracy of the tree based solution as the degree K varies for the accumulation of the same random input array. The accuracy is evaluated versus a quadruple precision execution which we assume (even if not fully deterministic) to be the most precise since the amount of rounding and truncation errors is minimized implicitly.

The tree based approach is a "left-leaning" tree where threads with lower rank do more work. The algorithm is not load-balanced but allows a "right-leaning" correction tree to be used to increase accuracy as presented in [8]. We plan to explore this possibility as future work.

5 Conclusions

We have investigated the effects of the problem of non-deterministic accumulation on the convergence of a conjugate gradient calculation used as part of a power grid analysis application (Power State Estimation (PSE)) on the Cray XMT system. We have investigated a library-based solution which performs the accumulation using a parallel, tree-based prefix scheme. Our library-based tree solution exhibits some overhead with respect to the XMT compiler-based solution, since it introduces a function call. Having a function call for each accumulation can inhibit powerful compiler transformations such as loop fusion, software pipelining, unroll-and-jam and others which have been shown to significantly improve performance. Moreover, in many cases, an accumulation is performed in a parallel loop that also executes other operations. Therefore, we motivate the need to integrate a precise accumulation algorithm at compiler level potentially using a new pragma (i.e. `#pragma mta precise reduction`). The new pragma could indicate where the programmer intends to have reductions executed with absolute deterministic behavior. The run-time and the compiler could then choose the right degree K based on thread and processor counts.

References

[1] IEEE Standard for Floating-Point Arithmetic. Technical report, 2008.

[2] A. Abur and A. Exposito. *Power System State Estimation: Theory and Implementation.* Marcel-Dekker, 2004.

[3] Y. Ben-Asher and G. Haber. Parallel solutions of simple indexed recurrence equations. *IEEE Trans. Parallel Distrib. Syst.*, 12(1):22–37, 2001.

[4] R. Brent and H. Kung. A regular layout for parallel adders. *Computers, IEEE Transactions on*, C-31(3):260–264, March 1982.

[5] R. Brightwell, B. Lawry, A. B. MacCabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 268, Washington, DC, USA, 2002. IEEE Computer Society.

[6] D. Chavarría-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer. Early Experience with Out-of-Core Applications on the Cray XMT. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, April 2008.

[7] J. Feo, D. Harper, S. Kahan, and P. Konecny. ELDORADO. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.

[8] N. Kapre and A. DeHon. Optimistic parallelization of floating-point accumulation. *Computer Arithmetic, IEEE Symposium on*, 0:205–216, 2007.

[9] J. F. Kruy. A fast conditional sum adder using carry bypass logic. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 695–703, New York, NY, USA, 1965. ACM.

[10] H. Leuprecht and W. Oberaigner. Parallel algorithms for the rounding exact summation of floating point numbers. Computing, vol. 28,, 1982.

[11] J. Nieplocha, A. Marquez, V. Tipparaju, D. Chavarría-Miranda, R. Guttromson, and H. Huang. Towards efficient power system state estimators on shared memory computers. In *Proceedings of the 2006 IEEE Power Engineering Society General Meeting*, 2006.

[12] K. Ozaki, T. Ogita, S. M. Rump, and S. Oishi. Fast and robust algorithm for geometric predicates using floating-point arithmetic. *Trans. JSIAM*, 4(4):2006.