

Investigation of Realistic Performance Limits for Tera-Scale Computations

T.A. Brunner and U.R. Hanebutte

This article was submitted to
High Performance Computing 2000
Washington, DC
April 16-20, 2000

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

September 9, 1999

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (423) 576-8401
<http://apollo.osti.gov/bridge/>

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161
<http://www.ntis.gov/>

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
<http://www.llnl.gov/tid/Library.html>

INVESTIGATION OF REALISTIC PERFORMANCE LIMITS FOR TERA-SCALE COMPUTATIONS *

Thomas A. Brunner[†] and Ulf R. Hanebutte^{††}

[†] University of Michigan
1906 Colley Building
Ann Arbor, MI 48109

^{††} Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Box 808, L-560
Livermore, CA 94551

Abstract

The two key factors affecting the performance of tera-scale computations are the parallel efficiency of the underlying algorithms, and the local performance on a single processor. In the past, most attention was given to parallel efficiency and parallel scalability. This led to algorithms and techniques that provide good scalability and parallel efficiency. However, it was often assumed that local computations, which require no inter-processor communications, could be performed at a high single processor performance rate (i.e. a high fraction of the advertised peak floating point arithmetic performance). For today's parallel computers, this might not be achievable. An investigation of realistic performance limits on a single processor is the focus of this paper.

1 Introduction

Tera-scale computations can be characterized as parallel computing on large number of processors (i.e., thousands of processors) with large local memory (i.e. gigabytes of local memory on each processing node). These computers exhibit a high aggregated theoretical peak performance, measured in Top/s ($= 10^{12}$ op-

erations per seconds). The computations themselves contain large number of unknowns (e.g. Reference [3] reports a calculation containing over 14 billion unknowns). Working with these large data sets requires a high input/output bandwidth and substantial secondary storage space. The complexity of tera-scale computations is high, due to the fact that coupled physical systems (e.g. fluid dynamics combined with chemical reactions) are being modeled. Such computations require modern algorithms which are scalable, rapidly convergent, and often adaptive. To accommodate these needs, "lean" algorithms have been developed that require fewer floating-point operations per iteration step and therefore are low in their computational intensity. The definition of computational intensity and its implication in terms of performance is discussed below. Also, the implementations of tera-scale applications need to focus on portability and reusability, since their lack can cause costly expenditures.

Taking the above described complexity of tera-scale computing into account, it would be over simplistic to evaluate the merit of such computations exclusively on its performance. Performance is commonly measured as the number of floating-point operations per seconds. However, the floating-point performance is an important measure and will be examined here. The performance of a parallel algorithm is calculated by multiplying the parallel efficiency by the number of processors and by the single proces-

*This work was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

```

for( i=0; i<n; i++ ) {
    x[i] = x[i] + c*y[i];
}
// 2 Ld, 2 Fop, 1 St

```

Figure 1: Code fragment: Triad operation with vectors of length n , where Fop is floating-point operation, Ld is load, and St is store

single processor floating-point rate. While the parallel efficiency is the subject of many papers, we focus on the investigation of realistic performance limits on a single processor. Single processor code optimization is not a new discipline and computer vendors provide guides such as [1] to help application developers. Due to recent developments in computer hardware design single processor optimization gained significance in parallel algorithm development. For example, Gropp et al. report in [5] their research on implicit CFD codes, and Toledo [10] studied sparse-matrix vector multiplication. In the following sections we 1) present the ASCI Blue-Pacific SST system, 2) discuss the general concept of computational intensity and the memory bottleneck, 3) introduce two kernel routines which are associated with a projection method, and 4) give performance results for various implementations of these kernels.

2 The ASCI Blue-Pacific SST System

The ASCI Blue-Pacific SST system at LLNL [13] is a three-sector machine with each sector housing 432 four-way SMP compute nodes for a total of 1,296 compute nodes. Each processor is a 332 MHz PowerPC 604e capable of two operations per clock cycle, i.e. 664 Mop/s. The system peak performance, taking compute and support nodes into account, is 3.9 Top/s. The compute nodes are outfitted with either 1.5 or 2.5 gigabytes of memory resulting in a total of 2.6 Tbytes for the entire system.

A mixed computing model which combines a distributed and a shared memory computing model is required. In general, the message passing interface MPI [4] is used to distribute, execute and control the parallel computation among processing nodes. For the second level of parallelism, which exploits the four processors on a node, a number of options are provided to the application developer. One can resort

to parallelized mathematics and engineering libraries, write explicit parallel shared memory programs by utilizing the Pthread library [9], as well as introduce this second level of parallelism implicitly through the use of OpenMP [12] pragmas. In the case of OpenMP, the compiler will augment the program automatically. OpenMP is supported on the ASCI Blue-Pacific system through the KAP/PRO toolset [11].

3 Computational intensity and the memory access bottleneck

The peak performance of a single processor of the IBM ASCI Blue-Pacific is 664 Mop/s; however, in order to achieve peak performance data has to be in the registers. If only a few operations per data item are performed, such as it is the case for many scientific computations, memory bandwidth between main memory and the processing units becomes the limiting performance factor. Quoting Roger W. Hockney [6] "If there is a memory access bottleneck, then the key program variable to consider is the computational intensity, f , which is defined as the number of arithmetic operations performed per memory transfer. Put another way, it is how intensely one computes with data once it has been received in the registers (or cache) within the arithmetic unit. If the computational intensity is high, ... the memory bottleneck is not seen. On the other hand, if the computational intensity is low then the time spent on data transfer dominates the calculation, and the performance of the computer is much less than that advertised on the basis of the arithmetic rate." Hockney classifies problems according to their computational intensity as belonging to the class of order $O(1)$, $O(\log n)$ or $O(n)$ problems. For example, the computational intensity of the triad operation (Fig. 1), i.e. daxpy or saxpy is $f = 2/3$, while a matrix multiply of $2n \times n$ matrices has $f = 2/3n$.

In order to study the memory bandwidth bound for the four-way SMP node we extended the STREAM benchmark [8]. STREAM is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels which have a computational intensity of order $O(1)$. It is specifically designed to work with data sets much larger than the available cache on any given system, so that the results are (presumably) more indicative of the performance of very large vector style applications. The benchmark is comprised of a set of operations. Included in this suite is the triad operation (Fig. 1), which is a very common operation in scientific computing. Therefore, the triad operation is studied here.

A shared memory parallel implementation utilizing the Pthread library [9] as well as an OpenMP [12] implementation (utilizing the KAP/PRO toolset [11]) of the triad operation are compared to the single processor version of the triad algorithm. For the triad operation, which requires 2 load, 2 floating point and 1 store operations, a surprisingly low number of 17.5 MFlop/s were measured when utilizing only one of the four processors on a SP node. An aggregated performance of 49.6 MFlop/s was obtained for the 4-way SMP on blocked data, while memory contention reduces the performance below the single processor limit if data is non-blocked. The results given in Table 1 show that for blocked data the OpenMP implementation provides the same performance as the Pthread implementation, while for the non-blocked case the Pthread version outperforms the OpenMP code.

Work assignment	Pthreads	OpenMP
Blocked	49.6	49.6
Stride of 4	12.6	9.2

Table 1: Performance numbers (given in MFlop/s) of the Triad kernel for a vector length of 40,000,000. Pthread and OpenMP code take advantage of all four processors of a SMP node. Utilizing only one processor results in 17.5 MFlop/s.

4 The P and Pplus Kernels

In this section we introduce two kernel routines which are derived from an actual tera-scale application. These kernels are associated with a projection method [2] (hence the names P and Pplus) that is utilized in a neutral particle transport solver [3]. To facilitate the study, both kernels were extracted from the actual code and were integrated into a separate test framework. To verify accurate kernel behavior in the test framework, execution times were compared between the actual code and the kernel code, as given in Table 2. While the actual code contains two different versions of each routine, denoted by P1, P2, P1_plus and P2_plus, only one is discussed here. The original implementation of the two kernels are given in Fig. 2 and Fig. 3.

	P1	P2	P1_plus	P2_plus
Actual code	3.210	5.180	3.070	4.940
Kernel	3.086	4.991	3.085	5.025

Table 2: Comparison between kernel and actual code performance (wall clock in seconds). IBM Xprofiler tool is used to determine actual code performance.

The two variables, *number of elements* and *number of segments*, are problem dependent. While *number of elements* can vary in a wide range, *number of segments* can only be one of few discrete values. The first modification to the original routines utilizes the IBM ESSL library [7]. The loops including the triad operations are replaced by calls to the optimized library routines *daxpy* and *dyax*. The third implementation of the kernels is a hand tuned version given in Fig. 6 and Fig. 7. In the tuned routines, loops are unrolled, data reuse is optimized and load/store operations are explicitly coded. For brevity, only one case, where the *number of segments* is equal to 4 is shown here. All kernels were compiled with the IBM C-compiler *xl*. The compile options were the following: `-O3 -qarch=ppc -qtune=604 -qunroll=4`.

5 Kernel Performance

The three implementations of the P routine are compared for varying vector lengths (size 1 to $50^3 =$

```

for( n=0; n<num_segments; n++ ) {
  y_seg = y + n*num_elements;
  a      = A[n];
  for( i=0; i<num_elements; i++ )
    y_seg[i] += a * x[i];
}

```

Figure 2: Code fragment of the original P routine

```

for( i=0; i<num_elements; i++ ) x[i] = 0.;

for( n=0; n<num_segments; n++ ) {
  y_seg = y + n*num_elements;
  b = B[n];

  for( i=0; i<num_elements; i++ )
    x[i] += b * y_seg[i];
}

```

Figure 3: Code fragment of the original Pplus routine

125,000). The number of segments is set to 4. The performance is given in terms of Mop/s (i.e. 10^6 operations per second) in Fig. 8. Along the kernel results, the STREAM benchmark result for the triad operation is given by a dotted line. The original version of the P routine approaches the memory bandwidth limit provided by the STREAM benchmark when the vector size is large. Utilizing the ESSL library provides better performance for a vector length larger than 100, while the best overall performance is achieved by the hand tuned kernel. For large vectors, a factor of two performance gain can be seen for the hand tuned version compared to the original kernel. To obtain results free of cache effects the test framework flushed the cache prior to each kernel execution, thereby ensuring that the kernels accessed data from main memory rather than cache.

Performance numbers for the Pplus routine are given in Fig. 9. The three implementations of the Pplus routine are compared for varying vector lengths (size 1 to $50^3 = 125,000$). The number of segments is set to 4. The performance is given in terms of Mop/s (i.e. 10^6 operations per second) in Fig. 9. Along the kernel results, the STREAM benchmark result for the triad operation is given by a dotted line. As before,

```

for( n=0; n<num_segments; n++ ) {
  y_seg = y + n*num_elements;
  a      = A[n];
  daxpy(num_elements, a, x, 1, y_seg, 1);
}

```

Figure 4: Code fragment of the P routine utilizing the IBM ESSL library

```

b = B[0];
y_seg = y;
dyax( num_elements, b, y_seg, 1, x, 1);

for( n=1; n<num_segments; n++ ) {
  y_seg = y + n*num_elements;
  b = B[n];
  daxpy( num_elements, b, y_seg, 1, x, 1);
}

```

Figure 5: Code fragment of the Pplus routine utilizing the IBM ESSL library

the results are free of cache effects. The asymptotic limit of original version of the Pplus routine is slightly above the memory bandwidth. Similar to the P kernel, utilizing the ESSL library provides better performance for a vector length larger than 100, while the best overall performance is achieved by the hand tuned kernel. For large vectors, data reuse in the hand tuned kernel clearly contributes to the three fold performance increase compared to the original code.

6 Conclusion

We have shown that, in order to derive realistic performance limits for tera-scale computations one needs to identify the computational complexity of critical code components. If the computational complexity is low, i.e. the number of floating-point operations per data item is low, than the performance of the computation is limited by the memory access bandwidth and not by the peak performance of the processing units. Current hardware trends indicate an ever widening gap between CPU and memory performance. Thus, algorithms with low computational

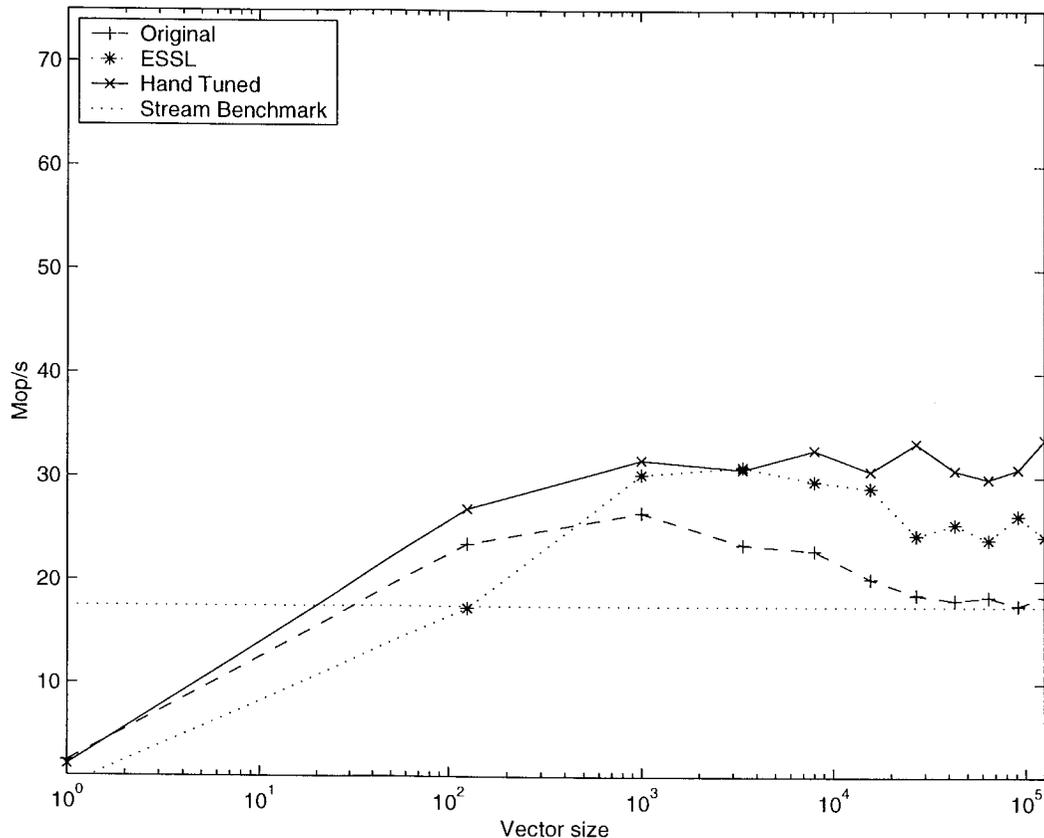


Figure 8: Performance numbers for the P routine

complexity achieve an ever declining fraction of peak.

Two representative kernels, derived from an actual tera-scale simulation were studied. These kernels represent algorithms with a low order of computational complexity. The original code and two alternative implementations are discussed and compared. Taking the memory bandwidth bound into consideration and aggressively restructuring to take advantage of data reuse and loop unrolling, a speedup of 2 to 3 times compared to the original implementation could be achieved.

References

- [1] Andersson, S., et al. *RS/6000 Scientific and Technical Computing: Power3 Introduction and Tuning Guide*. SG24-5155-00, <http://www.redbooks.ibm.com>
- [2] Brown, P.N. *A linear algebraic development of diffusion synthetic acceleration for 3-d transport equations*. SIAM J. Numer. Anal., 32 (1995), pp. 179-214.
- [3] Brown, P.N., Chang, B., Dorr, M.R., Hanebutte, U.R and Woodward C.S *Performing Three-Dimensional Neutral Particle Transport Calculations on Tera Scale Computers*. High Performance Computing '99 (part of the 1999 Advanced Simulation Technologies Conference), ISBN: 1-56555-166-4, page 76-81, April 11 - 15, 1999, San Diego, CA. also: UCRL-JC-132006
- [4] Gropp, W.D., Lusk, E., Skjellum, A. (1994). *Using MPI*. MIT Press, Cambridge, MA, 1994.
- [5] Gropp, W.D., Kaushik, D.K., Keyes, D.E. and Smith, B.F. *Towards Realistic Performance Bounds for Implicit CFD Codes*. Parallel CFD

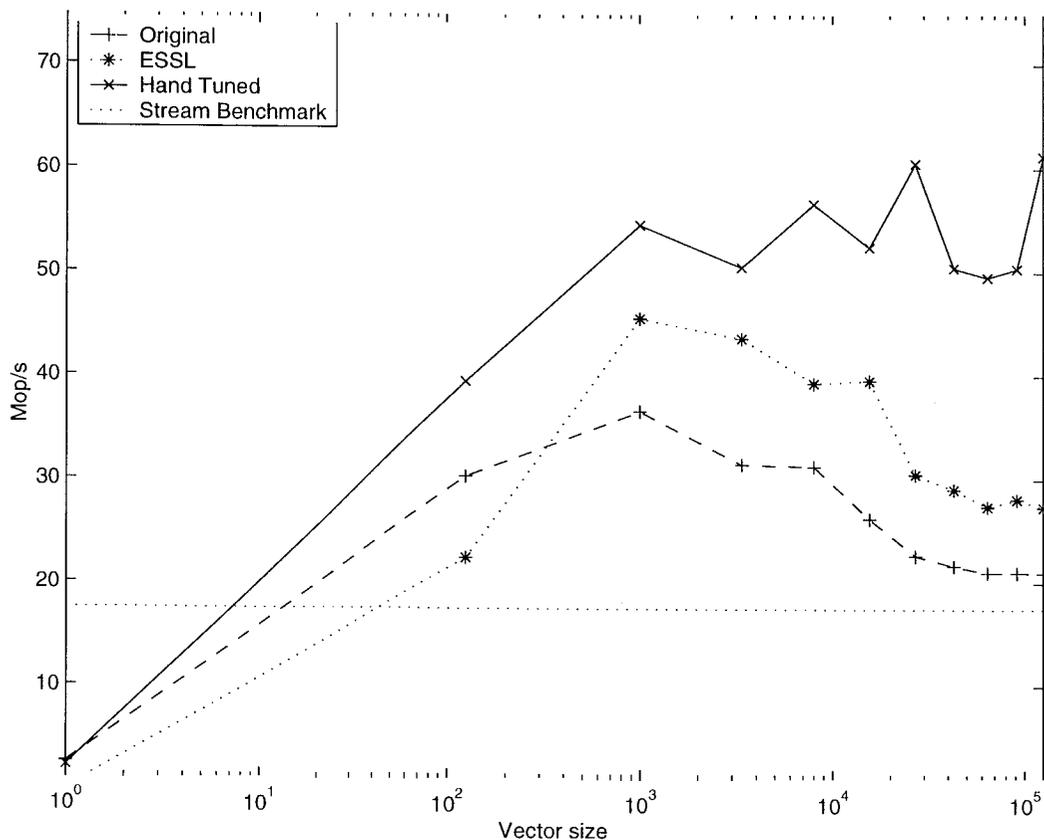


Figure 9: Performance numbers for the Pplus routine

- '99, May 23 - 26, 1999, Williamsburg, VA, proceedings to be published by North Holland
- [6] Hockney, R.W. *The Science of Computer Benchmarking*. SIAM, Philadelphia, PA, 1996.
- [7] IBM *Engineering and Scientific Subroutine Library for AIX*. Guide and Reference SA22-7272-01, http://www.rs6000.ibm.com/resource/aix_resource/sp_books/essl/
- [8] McCalpin, J.D. *STREAM: Sustainable memory bandwidth in high performance computers*. Technical report, University of Virginia, 1995. <http://www.cs.virginia.edu/stream>
- [9] Nichols, B., Buttler, D. and Proulx Farrell, J. *Pthreads Programming*. O'Reilly, 1996.
- [10] Toledo, S. *Improving the memory-system performance of sparse-matrix vector multiplication*. IBM J.Res. and Dev., 41: 711-725, 1997
- [11] www document *The KAP/Pro toolset for OpenMP*. <http://www.kai.com/parallel/kappro>
- [12] www document *Home page of the OpenMP Architecture Review Board*. <http://www.openmp.org>
- [13] www document *System Attributes for the ASCI Blue-Pacific Systems*. UCRL-MI-128208 <http://www.llnl.gov/asci/platforms/bluepac/blue.table.html>, January, 1, 1999.

```

switch(num_segments){
case 4:
    p0 = y ;
    p1 = y +   num_elements;
    p2 = y + 2*num_elements;
    p3 = y + 3*num_elements;
    a0 = A[0];
    a1 = A[1];
    a2 = A[2];
    a3 = A[3];
    for( i=0; i<num_elements; i++){
        aux = x[i];
        t0 = p0[i];
        t1 = p1[i];
        t2 = p2[i];
        t3 = p3[i];
        t0 = t0 + a0 * aux;
        t1 = t1 + a1 * aux;
        t2 = t2 + a2 * aux;
        t3 = t3 + a3 * aux;
        p0[i] = t0;
        p1[i] = t1;
        p2[i] = t2;
        p3[i] = t3;
    }
    break;
default:
    /* original code */
}

```

Figure 6: Code fragment of the hand optimized P routine

```

switch(num_segments){
case 4:
    p0 = y ;
    p1 = y +   num_elements;
    p2 = y + 2*num_elements;
    p3 = y + 3*num_elements;
    b0 = B[0];
    b1 = B[1];
    b2 = B[2];
    b3 = B[3];
    for( i=0; i<num_elements; i++ ){
        x[i] = b0 * p0[i] + b1 * p1[i]
              + b2 * p2[i] + b3 * p3[i];
    }
    break;
default:
    /* original code */
}

```

Figure 7: Code fragment of the hand optimized Pplus routine