Parallel Processing Letters
© World Scientific Publishing Company

NUMA-aware Multicore Matrix Multiplication

WAIL Y. ALKOWAILEET
Department of Computer Science (Systems),
University of California, Irvine, CA 92697, USA

DAVID CARRILLO CISNEROS
Department of Computer Science (Systems),
University of California, Irvine, CA 92697, USA

ROBERT V. LIM
Department of Computer Science (Systems),
University of California, Irvine, CA 92697, USA

ISAAC D. SCHERSON
Department of Computer Science (Systems),
University of California, Irvine, CA 92697, USA

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

A novel user-level scheduling, along with a specific data alignment method is presented for matrix multiplication in cache-coherent Non-Uniform Memory Access (ccNUMA) architectures. Addressing the data locality problem that occurs in such systems alleviates memory bottlenecks in problems with large input data sets. It is shown experimentally that a large number of cache misses occur when using an agnostic thread scheduler (such as OpenMP 3.1) with its own data placement on a ccNUMA machine. The problem is alleviated using the proposed technique for tuning an existing matrix multiplication implementation found in the BLAS library. The data alignment with its associated scheduling reduces the number of cache-misses by 67% and consequently the computation time by up to 22%. The evaluating metric is a relationship between the number of cache-misses and the gained speedup.

Keywords: ccNUMA, matrix multiplication, multicore, multi-socket.

1. Introduction

The first cache-coherent Non-Uniform Memory Access architecture was introduced by Advanced Micro Devices (AMD) in 2004 to replace the shared-bus (Front-Side Bus or FSB) [1]. Figure 1 shows the two architectures side-by-side. This architectural shift was necessary by the vendors to overcome the electrical limitation of increasing the bus frequency for higher granularity of processors. However, this shift has added another dimension of complexity to the programmer due to the different latencies of different memory accesses. For a memory and processor intensive computation, it may require the programmer to address the architectural differences before using any existing software packages. In this paper, we show the impact of such computation by evaluating an optimized and multithreaded BLAS (Basic Linear Algebra Subprograms) double-precision matrix multiplication routine (DGEMM) on such architecture. After analyzing the behavior of the DGEMM routine, we identify the bottlenecks and propose a solution that requires no modification in the underlying implementation to overcome those bottlenecks.

The rest of this paper is organized as follows: In Section 2, we explain briefly the architectural background of a dual-socket ccNUMA machine and the theoretical background of the matrix multiplication algorithms. In Section 3, we evaluate a general matrix multiplication routine (DGEMM) in an optimized Basic Linear Algebra Subroutines (BLAS) implementation. In Section 4, we detail our user-level scheduling and data alignment for matrix multiplication. In Section 5, we present the result of our experiment and the relationship between the cache-misses and gained speedup. In Section 6, finally, we presents our conclusion.

2. Background

2.1. *Dual-socket ccNUMA Architecture*

In a dual-socket architecture, each socket has an Integrated Memory Controller (IMC) to control the access to the main memory. Therefore, there are two different un-core (out of core) memory accesses. The first accessess the local memory through the local IMC. The second accesses remote memory by utilizing the remote IMC through the point-to-point interconnect (HyperTransport for AMD or QuickPath Interconnect for Intel) between the two sockets. Figure 4 shows the local and remote memory access.

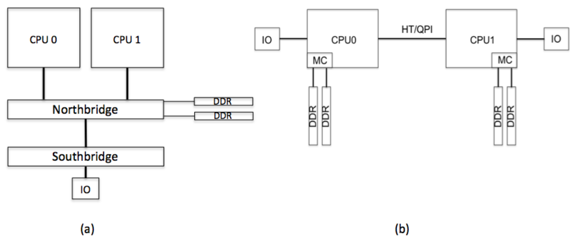

Fig. 1. (a) Shared-Bus Architecure. (b) ccNUMA Architecture.

For a SMP architecture, a protocol is required to maintain the coherency of the data in the distributed memories in addition to the distributed caches. Intel implements MESI[F] (Modified, Exclusive, Shared, Invalid, Forward) protocol in Intel Nehalem[a] processors to maintain the data coherency [2] [3]. In a multi-core processor, one of four un-core memory accesses will be initiated if a cache-line request missed the L1 and L2 caches: 1) local last level cache, 2) remote last level cache, 3) local DRAM or 4) remote DRAM, ordered by latency from lower to higher latency. When a Last Level (LL) cache-miss occurs and the cache-line is stored in the local memory address space, a request to the local IMC will be initiated to get the required cache-line. At the same time, a *snooping* [4] [1] [5] request to the remote cache will be initiated to check for a recent copy of the cache-line, Figure 2. On the other hand, if the required cache-line resides in the address space of the remote memory, a remote memory access will be initiated, Figure 3. Table 1 [5] shows the different latencies for the different types of un-core memory accesses in Intel Nehalem processor.

A study [6] targeted the data shuffling problem on similar architecture to maximize the overall performance of data shuffling applications (e.g sorting) in addition to the links throughput. In this study, we pursue a different objective where we want to minimize the inter-socket communication by improving the data locality of matrix multiplication routine.

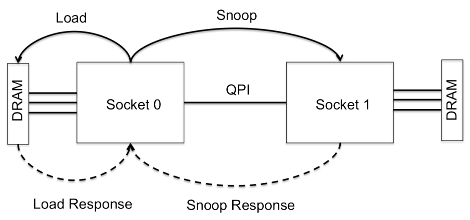

Fig. 2. Local memory access

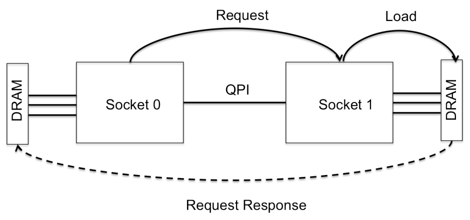

Fig. 3. Remote memory access

Fig. 4. cache-coherent Non-Uniform Memory Access.

Source	Cache-line State	Latency
L3 CACHE hit	Not Shared	40 cycles
	Shared in one of the lower level caches	65 cycles
	Modified	75 cycle
Remote L3 CACHE	-	100-300 cycles
Local DRAM	-	60 ns
Remote DRAM	-	100 ns

[a]We focus on Intel Nehalem in this paper as it is our experiment environment.

2.2. *Matrix Multiplication Asymptotic and Practical Complexities*

A study [7] has shown cache-oblivious matrix multiplications algorithms have the same cache-complexity as cache-aware algorithms for *ideal caches* [b]. Additionally, Prokop has shown that asymptotically faster algorithm (i.e Strassen's algorithm [8], which requires $O(N^{log_2 7})$ operations) has lower cache-complexity than the conventional divide-and-conquer algorithm, which requires $O(N^3)$ operations, for multiplying two square matrices. Table 2 compares the two algorithms' cache-complexities as in [7].

Algorithm	Asymptotic cache complexity
Straightforward[c]	$\Theta(N + N^2/L + N^3/L\sqrt{Z})$
Strassen	$\Theta(N + N^2/L + N^{log_2 7}/L\sqrt{Z})$

Different studies have shown that asymptotically faster algorithms are not always fast in practice. [9] [10] [11] and [12] have shown that for small matrices, the naïve 3-Nested-loops outperforms Strassen's algorithm. For larger matrices, a hybrid algorithm is suggested to compensate for the recursion overhead by fixing a cutoff point μ (or *Recursion Point* as in [12]). Therefore, when multiplying two square matrices A and B of size $N \times N$, the algorithm will run Strassen's recursively until $N \leq \mu$, then it will switch to the 3-Nested-loops algorithm. The cutoff point μ is architecture dependent and can be determined in a linear time. Finally, [12] has shown that exploiting data locality can be more beneficial for some architectures than reducing the asymptotic number of operations. In the next section, we evaluate an optimized matrix multiplication routine in BLAS (DGEMM) and relate those findings to our observation.

3. Parallel DGEMM Implementation Evaluation

A study has shown that UMA processors outperform ccNUMA ones in matrix multiplication. As a first attempt to see the effect of the ccNUMA architecture, we evaluated the DGEMM routine in Intel's BLAS implementation in Intel Math Kernel Library [13] using a dual-socket Quad-core Intel Nehalem processor. See Table 3 in Section 4 for more information about the experiment setup. At first, we used only one socket and 4 threads[d] to multiply two large matrices of size $N \times N$ where $N = \{8000, 12000, 16000, 20000, 24000, 32000\}$. In each run with different N, the DGEMM routine peaked at 36.1 GFLOPS (99.7% of the theoretical peak performance[14]). Running the same routine using the two sockets, got us only 56.0 GFLOPS of the 72.0 GFLOPS theoretical peak performance (22% less than the peak performance).

[b]Ideal caches are defined by the formula $Z = \Omega(L^2)$, Z is the cache size in words and L is the number of words in a cache-line

[d]No Hyper-threading.

To identify the bottlenecks of DGEMM routine multithreaded by OpenMP, we used Intel vTune [15] and Intel Performance Counter Monitor (PCM) [16] to understand the routine behavior of how threads were accessing the matrix elements. First, we monitored the QPI traffic using PCM. We found that the sockets exchange almost 8 GB/s. We suspected two possible problems that could explain the high volume of data exchange. First, there is a data locality problem where there is a high number of remote memory accesses. The second possibility is that there is a large number of LL cache-misses, where each requires snooping to the remote cache. Therefore, we profiled the number of LL cache-misses on both sockets. We found that as N increases, the number of cache-misses grows significantly. Figure 7 summarizes the experiment results.

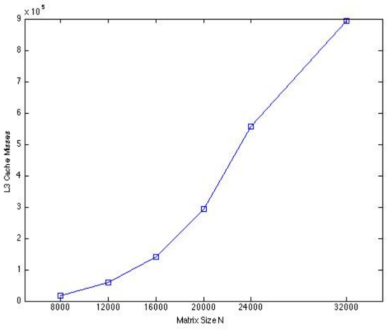

Fig. 5.

N	# of Cache misses
8000	16430
12000	58620
16000	141880
20000	293210
24000	556480
32000	893040

Fig. 6.

Fig. 7. LL cache-misses produced by multiplying two square matrices of size $N \times N$

4. NUMA-aware Matrix Multiplication

In this section, we detail our strategy to add NUMA-awaeness to matrix multiplication. This section shows how to 1) align the data and 2) schedule the threads for multiplying two matrices. We use the DGEMM routine provided by MKL as our low-level optimized matrix multiply implementation.

4.1. *Data Alignment*

BLAS matrix is represented as a one dimensional array. Figure 8 shows BLAS representation for a row-major matrix. [17] shows that certain matrix formats can reduce the number of cache-misses in a hierarchical memory. The study suggests wrapping the matrix elements into blocks where each block has its elements aligned in a Z-like shape (z-morton). In this work, we tile the two matrices A and B using the same transformation but with different blocks order. Figure 9 and Figure 10 show both matrices A and B tiled into blocks of size 2×2 respectively. The blocks order will be beneficial, as explained later in this section, to reduce the calculation required to compute the blocks' indices and to remove any dependencies between the blocks to compute the result matrix C.

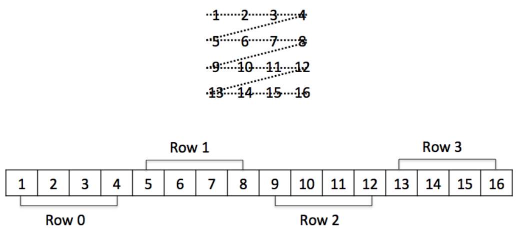

Fig. 8. BLAS row-major matrix of size 4×4

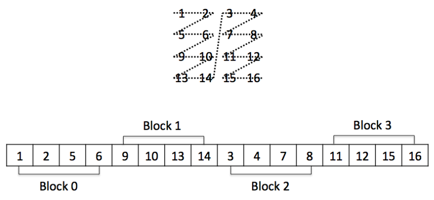

Fig. 9. Matrix A

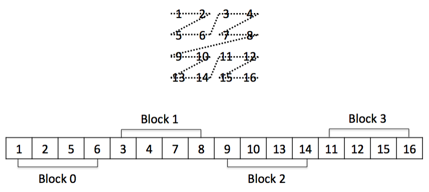

Fig. 10. Matrix B

Fig. 11. Tiled Matrices

4.2. *Threads Scheduling*

Our approach divides the available threads into two categories: Primary and Secondary threads. Primary threads (POSIX threads) prepare the the correspondent blocks of A and B before calling the matrix multiply routine. Secondary threads (OpenMP threads) are created after the blocks preparation to help the primary threads during the multiplication. The Primary-Secondary threads approach is targeted towards multiplying considerably large matrices. The size of N varies for different architectures and it can be determined in a linear search ($N \geq 4000$ for Intel Nehalem). One benefit of this approach is that it minimizes the calling overhead of the DGEMM routine. Additionally, it does not require any low-level tuning for any parameter (e.g block size), as we benefit from the DGEMM low-level optimizations.

Algorithm 1 shows the steps for each Primary-Secondary thread pair. In the case of dual Nehalem processors, there are 4 primary threads; 2 in each socket. Each primary thread prepares the same block of matrix A to be multiplied with a distinct block of matrix B to obtain a distinct result block of matrix C, Figure 12. Figure 13 shows an example of multiplying two square matrices tiled to 4 blocks. The main idea of our thread scheduling is to minimize retrieving the blocks from the main memory by keeping the block of matrix A and the blocks of matrix B in cache for as long as possible

Algorithm 1: 3-Nested-loop ccNUMA-aware matrix multiply algorithm

```
/* Do the outer-loop in parallel                                  */
foreach Thread t in PrimaryThreads do
    foreach Block blockA in Matrix A do
        for j ← 0 .. j ← B.NumberOfBlocks /
        PrimaryThreads.NumberOfThreads do
            /* Get the corespondent of Matrix B and Matrix C to the
               Thread t and blockA of Matrix A                    */
            blockB = B.getBlock(t.id, j, blockA) ;
            blockC = C.getBlock(t.id, j, blockA) ;
            /* Prepare the secondary thread                       */
            t.PrepareSecondaryThread() ;
            /* Do the blocked multiply using the primary and
               secondary threads. MKL DGEMM call                  */
            doBlockMultiply(blockA, blockB, blockC) ;
        end
    end
end
```

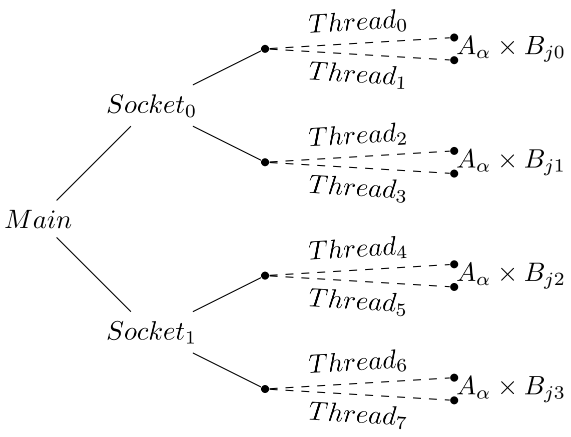

Fig. 12. Thread scheduling

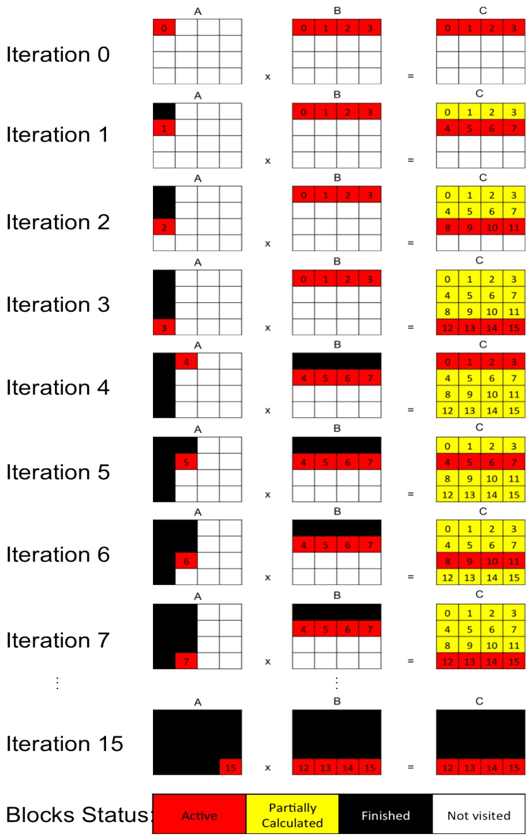

Fig. 13. Multiplying two matrices A and B of size 4 blocks using Algorithm 1

4.3. *Matrix Allocatoion*

Another important aspect that has to be taken into consideration is where to allocate the matrices among the distributed memories. Using only the *malloc* routine to allocate the space will end up placing the matrices in one memory (or two if the first is maxed out). This can lead to several problems like memory contentions and higher access latency for the remote socket. Therefore, a balanced allocation should be done to overcome such problems. In this work, we allocate the matrix A in socket's 0 memory and matrix C in socket's 1 memory. For matrix B, we distribute the blocks in an interleaved manner among the memories.

5. Experiment and Result

Table 3 summarizes the experiment environment setup. In this experiment, we are multiplying two square matrices of size $N \times N$, for $N = \{8000, 12000, 16000, 20000, 24000, 32000\}$, 10 times and measure the following:

- QPI traffic for one second interval.
- Number of cache-misses.
- Elapsed time.
- Floating-point Operations Per Second (FLOPS).

For the first measurement, Table 4 shows that our thread scheduling and the data alignment (MKL-NUMA) significantly reduced the inter-socket communication to one-third (on average compared with MKL-OpenMP). The correlation between the inter-socket communication reduction and the LL cache-miss rate of our strategy appears in Figure 14, which proves our premise about the data locality problem in MKL-OpenMP. Overcoming those issues may not give us the same percentage in the elapsed time to finish the multiplication, Figure 15. However, the gained speedup percentage of MKL-NUMA against MKL-OpenMP grows significantly as N increases, Figure 17. For $N = 32000$, the gained speedup reached 22% of the total execution time of MKL-OpenMP. Finally, we observed that the number of LL cache-misses of MKL-OpenMP grows in a way similar to that of the gained speedup percentage of our approach. After normalizing the two charts, Figure 19, we can see the relationship between the two measurements which gives us a better understanding of the nature of the memory effect and the projected speedup gain.

Processor	2 x Intel Xeon E5520 - Quad Core
QPI	2 x 5.86 GT (22 GB/s)
Memory Bandwidth	25.6 GB/s
OS	Ubuntu Server 12.04 LTS, kernel 3.5.0-23-generic
Compiler	GCC 4.6.3
Optimization Flag	-O2
OpenMP ver.	OpenMP 3.1
BLAS	MKL 11.0

N	MKL-OpenMP	MKL-NUMA
32000	7.34 GB/s	1.86 GB/s
16000	8.31 GB/s	1.51 GB/s
12000	8.46 GB/s	1.58 GB/s
8000	2.41 GB/s	1.57 GB/s

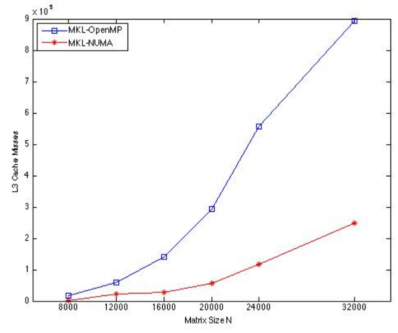

Fig. 14. # of LL cache-misses

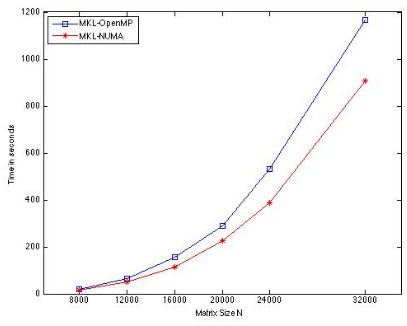

Fig. 15. Execution time

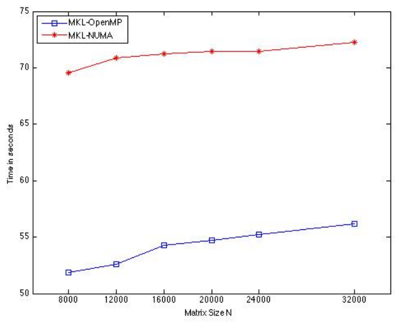

Fig. 16. Gflops

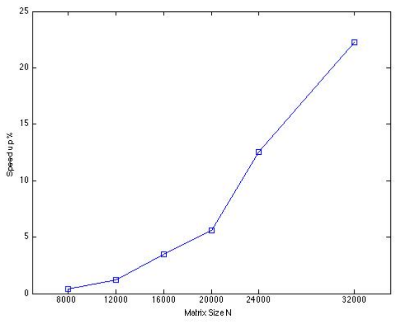

Fig. 17. Speedup %

Fig. 18. Experiment results

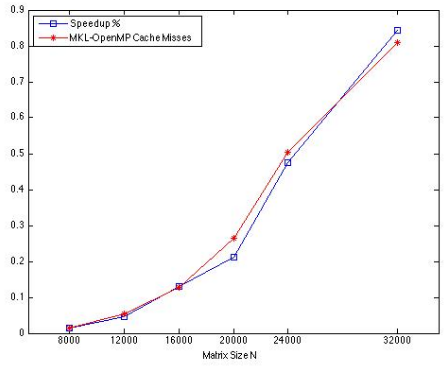

Fig. 19. The relationship between the gained speedup and the number of LL cache-misses in MKL-OpenMP

6. Conclusion

A NUMA-aware matrix multiplication implementation was presented. Simple user-level thread scheduling along with data alignment result in significant improvement towards utilizing memory hierarchy by addressing architectural awareness. Finally, we have shown through the relationship between the number of cache-misses and the gained speedup that our methods address the memory bottleneck issue.

References

[1] C.N. Keltcher, K.J. McGrath, A. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *Micro, IEEE*, 23(2):66–76, 2003.

[2] R.A. Maddox, G. Singh, and R.J. Safranek. *Weaving high performance multiprocessor fabric: architectural insights into the Intel QuickPath Interconnect*. Books by engineers, for engineers. Intel Press, 2009.

[3] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[4] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, September 1986.

[5] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 2009.

[6] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M Lohman. Numa-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[7] Harald Prokop. Cache-oblivious algorithms, 1999.

[8] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[9] R. P. Brent. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Numerische Mathematik*, 16:145–156, 1970.

[10] R. P. Brent. Algorithms for matrix multiplication. Technical Report TR-CS-70-157, Stanford University, Mar 1970.

[11] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 blas. *ACM Trans. Math. Softw.*, 16(4):352–368, December 1990.

[12] P. D'Alberto and A. Nicolau. Adaptive strassen and atlas's dgemm: a fast square-matrix multiply for modern high-performance systems. In *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*, pages 8 pp.–52, 2005.

[13] Intel Math Kernel Library reference manual. `http://download-software.intel.com/sites/products/documentation/doclib/mkl_sa/111/mklman.pdf`. Accessed: 2012-11-05.

[14] Intel xeon processor 5500 series. `http://download.intel.com/support/processors/xeon/sb/xeon_5500.pdf`. Accessed: 2013-06-20.

[15] Intel VTune Amplifier XE 2013 product brief. `http://software.intel.com/en-us/sites/default/files/Intel_VTune_Amplifier_XE_2013_PB.pdf`. Accessed: 2013-06-20.

[16] Intel performance counter monitor - a better way to measure cpu utilization. `http://www.intel.com/software/pcm`. Accessed: 2013-06-20.

[17] M. Thottethodi, Siddhartha Chatterjee, and A.R. Lebeck. Tuning strassen's matrix multiplication for memory efficiency. In *Supercomputing, 1998.SC98. IEEE/ACM Conference on*, pages 36–36, 1998.