IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 4, Ver. II (Jul-Aug. 2014), PP 21-26
www.iosrjournals.org

A Multi Core Hyper-Threaded Solution of a System of Linear Equations for Intelx64 Architecture

[1]Richa Singhal, [2]Dr. Sanjeev Bansal
[1]M-Tech C.S.E. Weekend
[2]Director, AMITY Business School

Abstract: *A system of linear equations forms a very fundamental principal of linear algebra with very wide spread applications involving fields such as physics, chemistry and even electronics. With systems growing in complexity and demand of ever increasing precision for results it becomes the need of the hour to have methodologies which can solve a large system of such equations to accuracy with fastest performance. On the other hand as frequency scaling is becoming limiting factor to achieve performance improvement of processors modern architectures are deploying multi core approach with features like hyper threading to meet performance requirements. The paper targets solving a system of linear equations for a multi core INTELx64 architecture with hyper threading using standard LU decomposition methodology. This paper also presents a Forward seek LU decomposition approach which gives better performance by effectively utilizing L1 cache of each processor in the multi core architecture. The sample uses as input a matrix of 4000x4000 double precision floating point representation of the system.*

I. Introduction

A system of linear equations is a collection of linear equations of same variable. A system of linear equations forms a very fundamental principal of linear algebra with very wide spread applications involving fields such as physics, chemistry and even electronics. With systems growing in complexity and demand of ever increasing precision for results it becomes the need of the hour to have methodologies which can solve a large system of such equations to accuracy with fastest performance. On the other hand as frequency scaling is becoming limiting factor to achieve performance improvement. With increasing clock frequency the power consumption goes up

$$P = C \times V^2 \times F$$

P is power consumption
V is voltage
F is frequency

It was because of this factor only that INTEL had to cancel its Tejas and Jayhawk processors. A newer approach is to deploy multiple cores which are capable to parallel process mutually exclusive tasks of a job to achieve the requisite performance improvement. Hyper threading is another method which makes a single core appears as two by using some additional registers. Having said that it requires that traditional algorithms which are sequential in nature to be reworked and factorized so that they can efficiently utilize the processing power offered by these architectures.

This paper aims to provide an implementation for standard LU decomposition method used to solve system of linear equations adopting a forward seek methodology to efficiently solve a system of double precision system of linear equations with 4000 variable set. The proposed solution addresses all aspects of problem solving starting from file I/O to read the input system of equations to actually solving the system to generate required output using multi core techniques. The solution assumes that the input problem has one and only one unique solution possible.

II. Challenges

The primary challenge is to rework the sequential LU decomposition method so that the revised framework can be decomposed into a set of independent problems which can be solved independently as far as possible. Then use this LU decomposition output and apply standard techniques of forward and backward substitution each again using multi core techniques to reach the final output.

Another challenge associated is cache management. Since a set of 4000 floating point variable will take a memory approximately 32KB of memory and there will 4000 different equations put up together, hence efficiently managing all data in cache becomes a challenge. A forward seek methodology was used in LU decomposition which tries to keep the relevant data at L1 cache before it is required to be processed. It also tries to maximise operations on set of data once it is in cache so that cache misses are minimum.

Impact

With a 40 core INTEXx64 machine with hyper threading the proposed method could achieve an acceleration of ~72X in performance as compared to a standard sequential implementation.

State Of The Art

The proposed solution uses state of the art programming techniques available for multithreaded architecture. It also uses INTEX ADVANCED VECTOR SET (AVX) intrinsic instruction set to achieve maximum hyper threading. Native POSIX threads were used for the purpose. Efficient disk IO was made possible by mapping input vector file to RAM directly using mmap.

Proposed Solution

A system of linear equations representing CURRENT / VOLTAGE relationship for a set of resistances is defined as

[R][I] = [V]

Steps to solve this can be illustrated as

a. Decompose [R] into [L] and [U]
b. Solve [L][Z] = [V] for [Z]
c. Solve [U][I] = [Z] for [I]

Resistance matrix is modelled as an array 4000x4000 of double precision floating type elements. The memory address being 16 byte aligned so that RAM access speeds up for read and write operations.

```
FLOAT RES[MATRIX_SIZE*MATRIX_SIZE]
__attribute__((aligned(0x1000)));
```

Voltage matrix is modelled as an array 4000x1 of double precision floating type elements. The memory address being 16 byte aligned so that RAM access speeds up for read and write operations.

```
FLOAT V[MATRIX_SIZE] _attribute__((aligned(0x1000)));
```

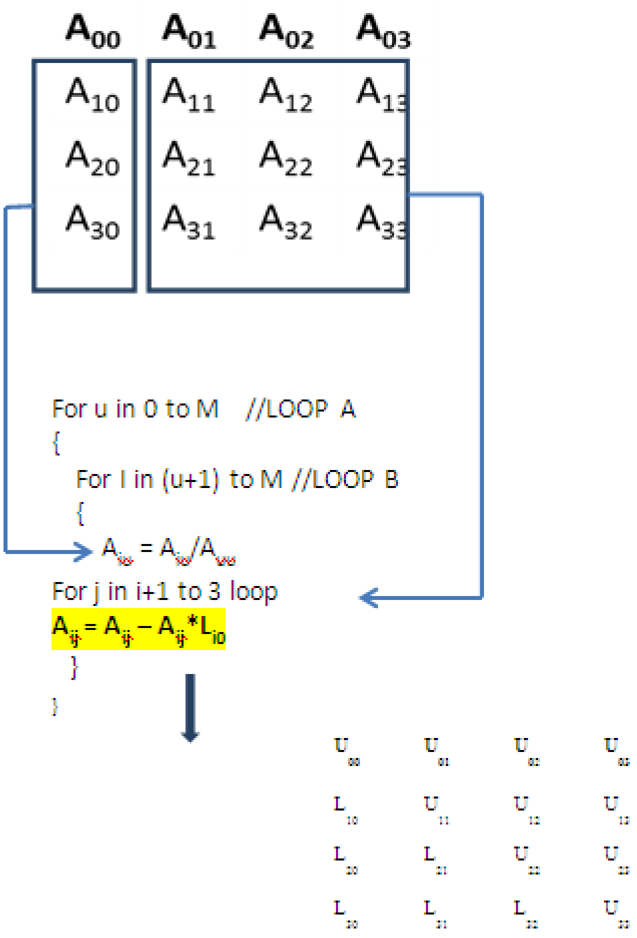
i. LU Decomposition

To solve the basic model of parallel LU decomposition as suggested above was adopted. Here as we move along the diagonal of the main matrix we calculate the factor values for Lower triangular matrix. Simultaneously each row operation updates elements for upper triangular matrix.

a. Basic routine to do row operation

This routine is the innermost level routine which updates the rows which will eventually determine the upper triangular matrix.

For each element of row there is one subtraction and one multiplication operation (highlighted).


A_{00} A_{01} A_{02} A_{03}
A_{10} A_{11} A_{12} A_{13}
A_{20} A_{21} A_{22} A_{23}
A_{30} A_{31} A_{32} A_{33}
U_{00} U_{01} U_{02} U_{03}
L_{10} U_{11} U_{12} U_{13}
L_{20} L_{21} U_{22} U_{23}
L_{30} L_{31} L_{32} U_{33}

```
For u in 0 to M   //LOOP A
{
  For I in (u+1) to M //LOOP B
  {
    $A_{iu} = A_{iu}/A_{uu}$
For j in i+1 to 3 loop
$A_{ij} = A_{ij} - A_{ij}*L_{i0}$
  }
}
```

LOOP B designates row major operation, while LOOP A designates column major operation.

Basic Algorithm

```
SUB LUDECOM (A, N)
  DO K = 1, n – 1
    DO I = K+1, N
      $A_{i, k} = A_{i, k} / A_{k, j}$
      DO j = K + 1, N
        $A_{i, j} = A_{i, j} - A_{i, k} * A_{k, j}$
      END DO
    END DO
  END DO
END LUDECOM
```

Each row major operation (LOOP B) iteration can be independently executed on a separate core. This was achieved by using POSIX threads which were non-blocking in nature. Because of mutual exclusion over the set of data MUTEX locks are not required provided we keep the column major operation (LOOP A) sequential.

Also for 2 consecutive elements in one row operation 2 subtraction and 2 multiplication operations are done. These 2 operations each are done in single step using Single Instruction Multiple Data instructions (Hyper threading)

Multi core Algorithm

```
SUB LUDECOM_BLOCK (A, K, BLOCK_START, BLOCK_END)
  DO I = BLOCK_START, BLOCK_END
```

```
    $A_{i, K} = A_{i, K} / A_{K, K}$
    DO j = K + 1, N
      $A_{i, j} = A_{i, j} - A_{i, K} * A_{k, K}$
    END DO
  END DO
END LUDECOM_BLOCK

SUB LUDECOM (A, N)
  DO K = 1, N – 1
    BLOCK_SIZE = (N – K) / MAX_THREADS
    Thread = 0
    WHILE (Thread < MAX_THREADS)
      P_THREAD (
          LUDECOMPOSITION_BLOCK (A,
          K,
          Thread*BLOCK_SIZE,
          Thread*(BLOCK_SIZE + 1)
        )
    ENDWHILE
  END DO
END LUDECOM
```

b. Forward substitution

Once LU decomposition is done, forward substitution gives matrix [Z]. Here again Single Instruction Multiple Data instructions are used

[L][Z] = [V] for [Z]

c. Backward substitution

After forward substitution final step of backward substitution gives current matrix [I]

[U][I] = [Z] for [I]

Here again Single Instruction Multiple Data instructions are used

Cache Improvements

On profiling it is observed that the core processing in above solution happens to be LU decomposition. However if we create threads equal in number to available cores the result was improving but not in same proportion to the number of cores. A VALGRIND analysis of cache performance reveals that because of large size of matrix each row operation was suffering a performance hit due to cache misses happening.

If we observe above solution it could be observed any $j^{th}$is processed for (j – 1) columns. So (j – 1) threads are forked for each iteration of column major operation (LOOP A). The data to be processed refers to same memory location but by the time next operation or thread is forked for the same row the corresponding memory data had been pushed out of lower level caches. Thus cache miss happens.

To solve this we adopted a forward seek approach wherein we first pre-process a set of columns sequentially thus enabling more operations on a row to be performed in the same thread. Now the data happens to be at lower level cache as we do not have to wait for another thread to process the same row.

Multi core Algorithm with forward seek operation

```
SUB LUDECOM_BLOCK_SEEK (A, K, S, BLOCK_START, BLOCK_END)
    DO I = BLOCK_START, BLOCK_END
      DO U = 1, S
        M = K + U -1
        $A_{i, M} = A_{i, M} / A_{M, j}$
        DO j = K + M + 1, N
          $A_{i, j} = A_{i, j} - A_{i, M} * A_{K, M}$
        END DO
      END DO
```

```
   END DO
  END LUDECOM_BLOCK

SUB LUDECOM (A, N)
  K = 1
  WHILE (K <= N)
   //Forward seek
   DO J = K, K + F_SEEK
     LU_DECOM_BLOCK_SEEK (A, J, 0, J, J+F_SEEK)
   END DO

   //Multi core
   K = K + F_SEEK
   DO L = 1, N –1
     BLOCK_SIZE = (N – L) / MAX_THREADS
     Thread = 0
     WHILE (Thread < MAX_THREADS)
       P_THREAD (
            LUDECOMPOSITION_BLOCK (A,
            L,
            F_SEEK,
            Thread*BLOCK_SIZE,
            Thread*(BLOCK_SIZE + 1)
          )
     ENDWHILE
   END DO
  END WHILE
END LUDECOM
```

III. Conclusion

Results

For purpose of computation a sample array of double precision floating point matrix of size 4000x4000 was taken. Performance numbers were generated on an 8 core INTEL architecture machine.

Experiment	Result
Simple linear program	200 Seconds
Program with 8 threads (2 Hyper threads in each thread)	~12 Seconds (APPROX. 16x faster)
Program with 72 threads on NEHALAM (64 cores) with forward seek	~1 second

TABLE 4.i

A programmer that writes implicitly parallel code does not need to worry about task division or process communication, focusing instead in the problem that his or her program is intended to solve. Implicit parallelism generally facilitates the design of parallel programs and therefore results in a substantial improvement of programmer productivity.

Many of the constructs necessary to support this also add simplicity or clarity even in the absence of actual parallelism. The example above, of List comprehension in the sin() function, is a useful feature in of itself. By using implicit parallelism, languages effectively have to provide such useful constructs to users simply to support required functionality (a language without a decent for() loop, for example, is one few programmers will use).

Languages with implicit parallelism reduce the control that the programmer has over the parallel execution of the program, resulting sometimes in less-than-optimal solution The makers of the Oz programming language also note that their early experiments with implicit parallelism showed that implicit parallelism made debugging difficult and object models unnecessarily awkward.[2]

A larger issue is that every program has some parallel and some serial logic. Binary I/O, for example, requires support for such serial operations as Write() and Seek(). If implicit parallelism is desired, this creates a new requirement for constructs and keywords to support code that cannot be threaded or distributed.

References

[1]. ^ Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.

[2]. **Jump up**^ S.V. Adve et al. (November 2008). "Parallel Computing Research at Illinois: The UPCRC Agenda" (PDF). Parallel@Illinois, University of Illinois at Urbana-Champaign. "The main techniques for these performance benefits – increased clock frequency and smarter but increasingly complex architectures – are now hitting the so-called power wall. The computer industry has accepted that future performance increases must largely come from increasing the number of processors (or cores) on a die, rather than making a single core go faster."

[3]. **Jump up**^ Asanovic et al. Old [conventional wisdom]: Power is free, but transistors are expensive. New [conventional wisdom] is [that] power is expensive, but transistors are "free"

[4]. Bunch, James R.; Hopcroft, John (1974), "Triangular factorization and inversion by fast matrix multiplication", Mathematics of Computation **28**: 231–236, doi:10.2307/2005828, ISSN 0025-5718.

[5]. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), Introduction to Algorithms, MIT Press and McGraw-Hill, ISBN 978-0-262-03293-3.

[6]. Golub, Gene H.; Van Loan, Charles F. (1996), Matrix Computations (3rd ed.), Baltimore: Johns Hopkins, ISBN 978-0-8018-5414-9.