On Supernode Transformations And Multithreading For The Longest Common Subsequence Problem

Johann Steinbrecher and Weijia Shang
Santa Clara University
500 El Camino Real
Santa Clara, CA 95053
j1steinbrecher@scu.edu, wshang@scu.edu

Abstract

The longest common subsequence (LCS) problem is an important algorithm in computer science with many applications such as DNA matching (bio-engineering) and file comparison (UNIX diff). While there has been a lot of research for finding an efficient solution to this problem, the research emphasis has shifted with the advent of multicore architectures towards multithreaded implementations. This paper applies supernode transformations to partition the dynamic programming solution of the LCS problem into multiple threads. Then, we enhance this method by presenting a transformation matrix that skews the loop nest such that loop carried dependencies of the inner loop are eliminated in each supernode. We find that this technique performs well on microarchitectures supporting out-of-order execution while in-order execution machines do not benefit from it. Furthermore, we present a variation of the supernode transformations and multithreading strategy which groups entire rows of the index set to form a supernode. The inter thread synchronization is performed by an array of mutexes. We find that this scheme reduces the amount of thread management overhead and improves the data locality. The techniques are benchmarked on a 12 core and a four core machine. At the 12 core machine the traditional supernode transformation speeds up the original loop nest 16.7 times. We enhance this technique to score a 42.6 speedup and apply our new method scoring a 59.5 speedup. We experience the phenomenon of superlinear speedup as the performance gain is larger than the number of processing cores. Concepts presented and discussed in this paper on the LCS problem are generally applicable to regular dependence algorithms.

I. Introduction

The LCS algorithm solves the problem of finding the LCS shared by two strings. It is well-known for its application at DNA matching in bio-engineering. Here, two DNA’s are represented as strings of characters ‘ACGT’. Finding out how similar two DNA’s are is important when researching the properties of new DNA’s or for criminal evidence. Furthermore, the LCS problem has its application in file comparison utilities such as the UNIX diff program, data compression, editing error correction and syntactic pattern recognition as well as the evidence of plagiarism (7, 15). The dynamic programming solution to the LCS problem is asymptotically bounded by $O(N^2)$ for $N$ character inputs. Considering the human DNA which is organized in 23 chromosomes each holding $50 \cdot 10^6$ to $220 \cdot 10^6$ base pairs, where each base pair is represented by a character, this generates a significant amount of computation. This paper discusses multithreaded implementation of the dynamic programming solution of the LCS problem to utilize the resources of multiple instruction multiple data machines (MIMD) as efficient as possible. We apply previous work on thread partitioning and supernode transformations to this problem, enhance these methods and present a variation of the supernode transformations and multithreading strategy. Concepts, ideas and observations made and presented in this paper on the LCS problem are generally applicable to regular dependence algorithms.

Copyright 2012, Australian Computer Society, Inc. This paper appeared at the 10th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia. January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 127. J. Chen and R. Ranjan, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.
instruction multiple data (SIMD) architectures such as most general purpose graphics processing units (GPGPU) and MIMD architectures such as many integrated cores (MIC) provide up to hundreds of cores. In embedded computing the trend clearly is towards parallel architectures, either by providing multiple cores inside the host processor or by building a heterogeneous system where the host processor is supported by a multi-core graphics processing unit. While supercomputers usually have physically distributed-memory, GPGPU’s and MIC’s have a centralized main memory shared by all cores. Cache memories may be distributed to different cores. For optimizing the processor utilization of each core, hyperthreading is used. Hyperthreading is a hardware technique supporting thread execution switch if the functional units are stalled by the currently running thread. These technological facts and trends motivate to understand how the LCS problem can be implemented to saturate the resources of a multi-core system.

This paper performs supernode transformations on the LCS problem in order to utilize the resources of a parallel system as efficient as possible. Hodzic and Shang presented in [10] supernode transformations for loop nests with regular dependencies. They group iterations in the index set to form supernodes. Those supernodes are then scheduled according to a linear schedule, which is respecting the data dependence, to the processing units. While this paper exploits parallelism amongst supernodes, it does not consider how to form supernodes in order to exploit the resources of each processing core. Therefore, we enhance this method by skewing the loop. We present a transformation matrix that eliminates the loop carried dependencies of the inner loop. This enables processing cores (hardware) or compilers (software) to reorder the instructions for hiding memory access latencies. Furthermore, we present a variation of the supernode transformations and multithreading strategy which groups entire rows of the index set to form a supernode. The inter thread synchronization is performed by an array of mutexes. We find that this scheme reduces the amount of thread management overhead and improves the data locality. We apply this method then to the original and skewed loop nest. The total execution time of each method is presented as a function of the computation time and communication time assuming systems with infinite resources and limited resources. The techniques presented and discussed in this paper are benchmarked on two multi-core MIMD machines with four cores and 12 cores, respectively.

The rest of the paper is organized as follows. In section II related work and our contribution is discussed. Section III presents architecture, programming and algorithm models and the concepts of linear scheduling. Section IV gives an overview over the basic ideas and implementation strategies. The total execution time at systems with limited resources is discussed in section V. In section VI our implementations are benchmarked on multi-core machines. Section VII concludes this paper.

II. Related work and our contribution

The LCS problem has been studied over the previous decades from different point of views. The dynamic programming solution was originally presented by Hirschberg in [9] performing the algorithm in an $O(N^2)$ complexity, where $N$ equals the size of the input strings. Based on this sequential method there have been many efforts for optimizing the time complexity, the space complexity, implement strategies using special data structures and design algorithms for special input properties targeting various systems as summarized in [4]. Our paper investigates on how to execute the problem on multi-core systems. Mabrouk presents in [3] a survey on the parallel complexity of the LCS problem on various target machines referring to papers in the time range from 1990 to 2006. Furthermore, this paper contributes a greedy strategy to schedule iterations on available machines in a parallel system. A summary of systolic algorithms and an efficient hardware-implementable systolic algorithm for the LCS is presented in [12]. While most commonly two strings are analyzed for their LCS, paper [13] discusses an optimal implementation on how to find the LCS of multiple input sequences (MLCS). It presents an efficient parallel algorithm for the MLCS based on a dominant points method. A coarse grained multithreaded implementation of the LCS is discussed in [7]. Supernode transformations for optimal running time on loop nests with regular dependencies are discussed in [10]. The supernode shape, expressed by the relative side length is discussed in [11] for optimal running time and discussed in [8] for minimizing the communication volume.

Having discussed related work we summarize here our contributions. While previous work on parallel implementations of the LCS problem did not consider supernode transformations we incorporate the concepts from [8], [10] and [11]. Techniques presented in this paper are based on several important observations we made. First, we find that skewing the loop nest, which eliminates inner loop dependencies, speeds up the algorithm on out-of-order execution cores while it does not gain a speedup on in-order architectures. Therefore, we enhance Hodzic’s method ([10]) by loop skewing. By benchmarking the technique on a 12 core MIMD machine with out-of-order pipelines we have found that the original supernode transformations speedup the LCS problem 16.7 times while our enhancement achieves a 42.6 times speedup over the original loop nest. As these performance gains are larger than the number of processing cores, we experience the
The LCS problem is a very data intense algorithm, reducing cache misses results in a large performance gain. This observation motivated us to design and present a variation of the supernode transformations and multithreading strategy scoring a speedup of $59.5$ on the 12 core machine. While paper [10] presents an expression for the total execution on systems with infinite resources we contribute a formula for the total execution time on systems with limited resources for Hodzic’s method and our new technique. While we find that the methods presented in this paper score phenomenal speedups, how to incorporate the concepts and ideas of paper [7] will be conducted in future work.

### III. Algorithm, architecture and programming models

In this section we present an overview on our algorithm model for the LCS problem, architectural model, programming model and linear scheduling.

#### A. Algorithm model

The LCS algorithm takes as inputs two sequences of characters: $x[1...M]$ and $y[1...N]$. The program then attempts to find the longest subsequence that is shared by these two strings, and returns the length of that subsequence. For example, for the input strings ‘ABCBDAB’ and ‘BDCABA’, the program will output four, because the longest common subsequence is ‘BCBA’, which has a length of four. The dynamic programming version of the LCS algorithm can be found in [6] and presented in figure 1.

The LCS problem is an algorithm with regular dependencies [17]. An algorithm with regular dependencies is one containing array references such that the dependencies remain constant from one iteration to the next. Such algorithms may be described by two parameters, the dependence matrix $D$ and the iteration space $J$. For the LCS, the dependence matrix equals:

$$D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

The iteration space is the set of all iteration indexing vectors and shown in figure 3. The process of establishing the dependence matrix and the iteration space is beyond the scope of this paper and can be found in [17]. Intuitively, each point in the iteration space corresponds to an iteration in the loop. An arrow between two points indicates a dependence between the two iterations. The dependence matrix $D$ corresponds to the loop carried write after read (WAR) in figure 1. For the correctness these dependencies must be respected.

This paper has an emphasis on the overall performance of an implementation of the LCS problem. Before discussing optimal thread partitioning and loop transformations we analyze the loop body. The code as shown in figure 1 consists of three if-statements. If-statements generate branch instructions which reduce the size of basic blocks in the compiled code. This has a negative impact on the performance as branches usually require the hardware pipelines to stall. Hardware techniques such as speculation and branch prediction reduce this effect but can’t eliminate it. Therefore, we try to reduce the number of branches in the loop body. By analyzing the data dependencies we find the second and third if-statement to hold loop carried dependencies while the first if-statement does not. Therefore, we move the first if-statement out to a separate loop. The resulting code is shown in figure 2 and used throughout this paper. Furthermore, all codes and examples used in this paper assume a squared index set.

#### B. Linear schedule and wavefronts

A linear schedule $f$ (II) is a linear or affine function that maps multiple dimensional iteration vectors in the iteration space onto a set of execution times represented by integers. Multiple independent iterations are assigned the same execution time for parallel processing. How to identify linear schedules that respect the data dependencies and minimize the total execution time is discussed in [17]. For
the dependence structure of the LCS problem in figure 3, one feasible linear schedule that respects dependencies is \( f((i \ j)^l) = (1 \ 1)(i \ j)^l = i + j \).

The vector \((1 \ 1)\) specifying the linear schedule is called linear schedule vector and is denoted as \( \Pi \). The iteration space is partitioned by the linear schedule into a series of hyperplanes \((1 \ 1)\) \((i \ j)^l = \text{constant}\). These hyperplanes are called wavefronts that are perpendicular to \( \Pi \) (see [17]). Such wavefronts may be generated using CLooG [2], a loop transformation and code generation tool based on the polyhedral model. One observation is that all iterations in the same wavefront are independent.

C. Architecture and programming model

For the purpose of this paper, a computer system is modeled by the parameters \( p \) and \( t \) where \( p \) is the number of cores in the system and \( t \) is the number of hyper threads each core accommodates. We assume multiple instruction multiple data architectures (MIMD) with shared memory. MIMD architectures can be programmed amongst others with Open MP [5], Intel®/Cilk™Plus or Posix Threads. In this paper the Posix Thread programming model [1] is used. The main memory is shared, while each core may have its own cache. This stands in many aspects in contrast to the CUDA programming model and architectural model, where a single instruction multiple data (SIMD) architecture is programmed. One major limitation of CUDA is that only threads within the same thread block can communicate with each other efficiently. One of the advantages of CUDA is an extremely low thread creation overhead and cache access latency. The programming model in this paper is especially applicable to future MIC systems ( [14] and [16]). MIC’s can be classified as MIMD systems with SIMD and superscalar properties. The system consists of multiple cores running independent instruction streams (MIMD). Each core accommodates several functional units (superscalar) as well as a SIMD unit for fast floating point calculations.

IV. Basic ideas and concepts

The basic ideas and concepts for speeding up the dynamic programming algorithm of the LCS problem are illustrated in this section. Subsection IV-A shows how to partition the algorithm according to Hodzic’s method. In subsection IV-B, we enhance Hodzic’s method by skewing the loop. In section IV-C we propose a variation of the supernode transformations and multithreading strategy. Subsection IV-D discusses the proposed techniques.

A. Traditional supernode transformation and thread partitioning

Hodzic presents in paper [10] a supernode transformation for algorithms with regular dependencies. It discusses the supernode transformation and optimal grain size to minimize the total running time. Furthermore, in [8], [10] and [11] the supernode shape is discussed for minimum running time and minimum communication volume. The total running time is the sum of the computation time and the communication cost. Hodzic groups a number of computation nodes to form a supernode and assigns each supernode to a processor as an unit for execution. Applying this concept to our programming model we spawn and terminate a thread for each supernode. Hodzic’s paper uses a loop nest in Example 2.1 with the dependence matrix \( D \):

\[
D = \begin{pmatrix}
0 & 1 & 1 \\
1 & 0 & 1 \\
\end{pmatrix}
\]

As we have shown in section III-A this is the dependence matrix of the LCS problem. Therefore, we can apply the method as presented by Hodzic where the index set is partitioned into rectangular supernodes as shown in figure 4. Supernodes are formed by rectangles with the side length \( w \) and \( h \). The wavefronts for linear schedule \( \Pi \) (1 1) are indicated by transparent diagonal lines. In this technique, each thread executes one supernode. Assuming a system with infinite resources, all superiterations that share the same wavefront can be executed simultaneously on different cores. Therefore, a closed form expression for the total execution time is presented in section 3 of paper [10]:

\[
T = P \cdot (T_{\text{comp}} + T_{\text{comm}})
\]

Where \( P \), the number of computation phases, equals the linear schedule length or the number of wavefronts, \( T_{\text{comp}} \) equals the computation time per supernode and \( T_{\text{comm}} \) equals the communication time. Starting with this technique we will exploit further optimizations. While the actual computation time per supernode is strongly dependent on the input strings, this equation assumes that each computation phase takes the same time to execute. Furthermore, the aspect of smaller supernodes at the borders of the index set is not considered. We keep these
assumptions in all equations presented in this paper.

Fig. 4. The loop nest according to Hodzic’s supernode transformations (paper [10]).

One observation from this method is that while this linear schedule exploits parallelism among superiterations, the iterations within a supernode may hold dependencies. On out-of-order microarchitectures such as most recent x86 and ARM CPU’s, this may not perform best as instructions cannot be reordered for hiding memory access latencies because of dependence. This observation is the basis for subsection IV-B. Also, this method requires in our programming model to spawn and terminate a thread for each supernode. We will present a multithreading strategy that requires a reduced amount of thread management overhead, with an improved data access pattern for reducing cache misses, scoring the same amount of parallelism in subsection IV-C.

B. Eliminating inner loop carried dependencies

Mabrouk presents in section 2 of [3] how to transform the index set of the LCS in order to eliminate loop carried dependencies of the inner loop. Using the loop transformation and code generation tool ClooG, we experimented with this idea. The LCS is a very data intensive algorithm. Each computation node does at least three loads \((x[i], y[j], c)\) and one store. Therefore, the bottleneck of the algorithm are the memory access latencies. For minimizing stalls in the pipelines due to memory access latencies, two approaches are possible: hiding the latencies by loading several data in a pipelined fashion and reducing memory access latencies by exploring data locality for minimizing cache misses. In this section we investigate on hiding the memory access latencies by reordering the instructions. When skewing the loop nest as presented in [3] and shown in figure 5, there are no loop carried dependencies along the inner loop. This enables the software, i.e., the compiler or the hardware to reorder the instructions. One advantage is that multiple data can be loaded in a pipelined fashion to hide their latencies.

Figure 5 shows the skewed loop nest. This transformation can be formally described by the transformation matrix \(TR\):

\[
TR = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}
\]

The new dependence matrix is indicated by arrows in figure 5 and can be calculated by

\[
D_1 = TR \cdot D = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 2 \end{pmatrix}
\]

To support this observation we do a preliminary experiment. In figure 6 and figure 7 we show the results of executing the original and skewed loop nest on two out-of-order microarchitectures (x86-64) and one in-order microarchitecture (x86-64): an Intel® Atom™ D510 (Atom microarchitecture; in-order execution; 1.66GHz), an Intel® Core™ 2 U7300 (Core microarchitecture; out-of-order execution; 1.30GHz) and an Intel® Core™ i5-2500 (Sandy Bridge microarchitecture; out-of-order execution; 3.3GHz). Using GCC 4.6.1, we compiled the original loop nest and the skewed loop nest with an input size of 20000 characters per string performing no compiler optimizations (figure 6) and performing compiler optimizations (figure 7). Overall the skewed loop nest performs better on out-of-order execution microarchitectures while the in-order machine didn’t show any speedup. Noticeable is also that compiler optimization performed well on the in-order execution machine while it increases the execution time on the out-of-order architectures.

Based on this observation we enhance the strategy of Hodzic. Applying Hodzic’s supernode transformations on the skewed loop nest keeps the communication time constant while the computation time decreases due to the fact discovered in the experiment. This strategy is visualized in figure 8. Here the inner loop which iterates along the horizontal axis doesn’t have loop carried dependencies. Skewing the loop nest transforms also the dependence vectors as indicated by arrows in figure 8. Assuming a
speed-up due to loop skewing of $s$, the total execution time results in:

$$T = P \cdot \left( \frac{T_{\text{comp}}}{s} + T_{\text{comm}} \right)$$

C. Variation of the supernode transformation and multithreading strategy

The original implementation according to Hodzic has the disadvantage of grouping iterations with loop carried dependencies along the inner loop. We eliminated this by the method presented in section IV-B. One remaining down side of this technique is the high amount of thread creation overhead. For each supernode a thread is spawned and terminated. Generally, this can be reduced by partitioning the loop nest into as many supernodes as cores available in the system. This results in a minimum amount of thread management overhead but may not score a high degree of parallelism. Furthermore, the original supernode transformation does not optimize cache sharing among threads. The method presented in this subsection evolves from considering these aspects.

This method partitions the index set in vertical and horizontal zones of size $w$ and $h$. While previous methods scheduled each supernode in terms of a thread to a processor, this method executes one horizontal zone per thread. Each group of $w \times h$ index nodes is dedicated to a mutex. Therefore, the mutexes form a $[N/w] \times [N/h]$ matrix $M$ where $N$ is the dimension of the index set. This idea is illustrated in figure 9.

Here, the index set is divided in four horizontal zones $0,...,3$ and four vertical zones $0,...,3$ of size $w=2$ and $h=2$. The inter thread synchronization is performed by locking and unlocking the mutexes, where $M[k][l]$ corresponds to the mutex for vertical zone $k$ and horizontal zone $l$. Using the example from figure 9 we illustrate the method thoroughly. Initially, $Thread_0$ is created. Having finished with vertical zone 0 it creates $Thread_1$. Every thread is executing the same code. Therefore, in general $Thread_i$ creates $Thread_{i+1}$. To enforce the dependence structure of the loop nest, $Thread_k$ performing vertical zone $k$ locks $M[k][l]$ when entering horizontal zone $l$ and unlocks $M[k][l]$ when leaving horizontal zone $l$. Furthermore, after its creation $Thread_k$ executing vertical zone $k$, locks all mutexes $M[k+1][i]$ with $i=0,...,3$. Having finished horizontal zone $l$, $Thread_k$ unlocks mutex $M[k+1][l]$. To control the number of active threads $Thread_i$ waits until $Thread_{i-n}$ terminates, where $n$ is the number of threads that should be active at any time. According to this scheme the total execution time can be expressed as:

$$T = \left( \frac{N}{w} - 1 \right) \cdot T_{\text{zone}} + \left[ \frac{N}{h} \right] \cdot T_{\text{zone}},$$

where $T_{\text{zone}}$ is the amount of time required for each thread to execute one $w \cdot h$ group. Depending on $h$, this scheme may require a minimum number of thread creation.
Furthermore, a large number of horizontal zones may make it possible that cache lines, loaded by Thread\textsubscript{i}, can be reused by Thread\textsubscript{i+1}.

The technique as presented in figure 9 did not eliminate the loop carried dependencies. Therefore, we present the same idea on the skewed loop nest in figure 10. By this method we expect to decrease \( T\text{\_zone} \), one parameter of the total execution time.

![Fig. 10. Variation of the supernode transformation and multi-threading strategy on the skewed loop nest](image)

**D. Analysis and discussion**

In this subsection we analyze the advantages and challenges of the three methods presented. Applying Hodzic’s method to the LCS problem gave a general idea on how to partition the loop nest and how to schedule the supernodes to the processing units of a multi-core machine. We enhanced this method by loop skewing for reducing the computation time of each supernode (subsection IV-B). While this method improves the performance on architectures supporting out-of-order execution, in-order execution machines do not benefit from it. Furthermore, both methods generate an unnecessary high amount of thread control overhead as each supernode requires to spawn and terminate a thread. Our supernode transformation and threading strategy presented in subsection IV-C incorporates these observations. Furthermore, it provides an improved data access pattern. As Thread\textsubscript{i} executes the horizontal zone \( j \) of vertical zone \( i \) the data of this zone is loaded into the cache. Once Thread\textsubscript{i} has left zone \( j \), Thread\textsubscript{i+1} can enter it and read some of the required data from the cache instead of paying the penalty for accessing the main memory. Therefore, the sizes of the horizontal and vertical zones \( w \) and \( h \) are a function of the cache size and cache associativity. This method can be applied to the skewed loop nest as well. The technique is suitable for in-order and out-of-order execution architectures. We especially see potential for this method at future MIC architectures where a significant amount of processors and hardware threads are provided [14]. Such architectures usually consist of in-order execution architectures such as the Larrabee architecture [16].

**V. The total execution time on systems with \( p \) processing cores**

The total execution times used and presented in section IV assumed an infinite number of computation cores. This section enhances Hodzic’s formula used in sections IV-A and IV-B and the formula for our new methods presented in section IV-C for the case of \( p \) processors.

**A. Hodzic’s total execution time**

Hodzic presented the total execution time as the number of phases \( P \), which he defines as the number of wavefronts, multiplied with the sum of the computation time \( T\text{\_comp} \) and communication time \( T\text{\_comm} \).

\[
T = P \cdot (T\text{\_comp} + T\text{\_comm})
\]

While this remains generally true, the number of computation phases needs to be redefined. If all supernodes of a wavefront can be executed simultaneously, each wavefront requires one computation phase. If there are not enough resources in the system, a wavefront may require more than one computation phase. The total number of computation phases for this case can be estimated as:

\[
P = \sum_{i=1}^{W} (\max\left(\frac{w}{p}, 1\right))
\]

where \( p \) equals the number of processors, \( W \) equals the number of wavefronts and \( w_i \) equals the number of supernodes on wavefront \( i \).

**B. Total execution time of the new supernode transformation**

In section IV-C we presented the total execution time as

\[
T = (\left\lceil \frac{N}{w} \right\rceil - 1) \cdot T\text{\_zone} + \frac{N}{h} \cdot T\text{\_zone}
\]

We construct the execution time for \( p \) processors by considering the example in figure 11. Assuming \( p = 4 \) we execute four threads Thread\textsubscript{i} with \( i = 0, ..., 3 \). The execution order is illustrated in figure 12 by staggered horizontal lines.

Here the thread executing the very last horizontal zone is Thread\textsubscript{3}, which requires \( (\left\lfloor \frac{N}{w} \right\rfloor \mod 4) - 1 \cdot T\text{\_zone} \) due to dependencies to start and executes in \( (\left\lfloor \frac{N}{w} \right\rfloor / 4) \cdot \left\lfloor \frac{N}{w} \right\rfloor \cdot T\text{\_zone} \). Therefore, the total execution time results for the general case in

\[
T = (\left\lfloor \frac{N}{h} \right\rfloor \mod p) - 1 + \left(\left\lfloor \frac{N}{p} \right\rfloor \cdot \left\lfloor \frac{N}{w} \right\rfloor \right) \cdot T\text{\_zone}
\]
We benchmark our techniques on two out-of-order multi-core x86-64 machines, an Intel® Xeon® X5670 system with 12 cores and 24 threads ($p = 12$, $t = 2$ @ 2.93 GHz; a Nehalem architecture) and an Intel® Core™ i5-2500 with four cores and four threads ($p = 4$, $t = 1$ @ 3.3 GHz; a Sandy Bridge architecture).

A. The benchmarks

We implemented each technique discussed and presented in this paper in ANSI C using Posix Threads. The benchmark performances are compared against each other and against the unmodified original loop nest according to the code in figure 2. The input size of all benchmarks is 43500 characters randomly generated stored on the heap 43500 and unmodified across all benchmarks. Four methods are tested: Hodzic’s method (as shown in figure 4), Hodzic’s method enhanced (as shown in figure 8) and our variation of the supernode transformation and threading strategy applied to the original loop nest as shown in figure 9 and applied to the skewed loop nest as shown in figure 10. Implementations based on Hodzic’s method are benchmarked using squared supernodes as those are optimal for a squared index set according to [11]. For avoiding false sharing, mutexes which are stored globally are padded. Each benchmark is implemented with a maximum of four active threads on the four core machine ($p = 4$, $t = 1$) and 24 active threads on the 12 core machine with 24 hardware threads ($p = 12$, $t = 2$).

To measure the execution time the gettimeofday function is used. This kernel routine takes use of the virtual dynamically-linked shared objects library (VDSO) of the linux kernel. Therefore, no switching from user space to kernel space is required, which minimizes the measurements overhead. The benchmarks were compiled using GCC 4.6.1 and tested on linux platforms with kernel version 2.6.35. The compilation process took advantage of all compiler optimizations by using flag –O3. If the unoptimized code (compiler flag –O0) performed better than the optimized code such as in figures 13 and 14 we show those results as well. Each value in figures 13, 14, 16, 17, 18 and 19 represents the arithmetic average of four repetitive runs. All values in tables 15, 16, 17, 18 and 19 represent the execution time in seconds.

B. The results

Figures 13 (four core machine) and 14 (12 core machine) show the benchmarks of Hodzic’s method and our enhancement of it as described in subsection IV-A and subsection IV-B, respectively. The execution time is presented as a function of the supernode dimension, which is the side length
length of the squared supernodes. In both cases the curves share the general shape with which the execution time decreases with the supernode dimension until it reaches an optimal grain size. The fastest execution time in each graph is scored by applying Hodzic’s method to the compiler optimized skewed loop nest. The fastest execution time of our enhancement to Hodzic’s method in relation to the fastest execution time of Hodzic’s original supernode transformation scores a speedup of 3.01/1.80 = 1.67 at the four core machine and 1.82/1.09 = 1.67 at the 12 core machine. The optimal tile size for the compiler optimized transformed loop nest equals 500 at the four core machine and 260 at the 12 core machine. Interestingly is also that execution times for tile sizes between 230 and 500 are consistently low at the compiler optimized transformed curve in figure 13 (4 core) while this benchmark in figure 14 (12 core) shows a more distinct optimal tile size.

Fig. 15. Execution times of the original loop nest

Figure 15 shows the execution time of the original loop nest at each machine as presented in figure 2. Comparing the best performance of the traditional supernode transformation at the four core machine (i5core architecture) with the original loop nest, it shows a speedup of 51.33/3.01 = 17.1 and a 51.33/1.80 = 28.5 times speedup at the enhancement of Hodzic’s method. The benchmarks for the 12 core machine (Xeon architecture) score a 46.66/2.79 = 16.7 (traditional supernode transformation) and 46.44/1.09 = 42.6 (enhanced supernode transformation) speedup. These results are better than expected. We experience the phenomenon of super-linear speedup, where the speedup is larger than the the increase of computation cores. This is mainly due to reduced data access times. As one thread is performing on specific data the whole cache line is loaded. Threads may be able to use data that is still present in the cache as it got loaded by other threads for reducing cache misses.

Figures 16 and 17 present the results of our variation of the supernode transformation and threading strategy on the four core machine applied to the original and the skewed loop nest. Figures 18 and 19 show the same techniques on the 12 core machine. We present the results in tabular form as we experiment with two parameters: The number of horizontal zones $N/w$ and the number of vertical zones $[N/h]$. The results show that the execution time decreases with an increasing number of vertical zones and increases after it reached an optimal number of vertical zones. Furthermore, the benchmarks tend to perform better with increasing number of horizontal zones. The fastest execution time for each number of vertical zones is highlighted bold. While we experienced at the 12 core machine performance increases at larger numbers of horizontal zones we found that the four core machine has its performance optimum at lower numbers of horizontal zones. The data ranges in tables 16, 17, 18 and 19 present the critical subset out of all tested parameters. The new supernode transformation applied to the original loop nest scores a speedup of 51.33/3.17 = 16.2 on the four core machine and 46.44/0.79 = 58.8 on the 12 core machine.
over the original loop nest. The same method applied to the
skewed loop nest achieves a 51.33/1.76 = 29.2 speedup
on the core machine and a 46.44/0.78 = 59.5 speedup
on the 12 core machine. One observation is that skewing
the loop nest with this method improved the performance
only at the four core machine.

VII. Conclusion

This paper emphasized on supernode transformations
and multithreading for the LCS problem by applying
and enhancing previous work as well as presenting a
variation of the supernode transformation and threading
strategy. One major observation of this paper is that
multithreaded implementations of the LCS score super-
linear speedups. Applying Hodzic’s method (traditional
supernode transformation) to the LCS problem, it scores
a 16.7 times speedup over the original loop nest on a 12 core
MIMD machine. Enhancing this technique to eliminate
loop carried dependencies along the inner loop within each
supernode scored a 42.6 times speedup. To reduce the
thread management overhead and improve the data reusage
of threads we introduced a variation of the supernode
transformation and threading strategy scoring a 59.5 times
speedup. For each method we cited and presented the
functions for the total execution time consider systems
with infinite and systems with limited resources. The tech-
niques presented in this paper are especially applicable to
MIMD architectures. We benchmarked the techniques on
two modern x86 – 64 multi-core machines with four cores
and 12 cores. Techniques, ideas and formulas presented in
this paper on the LCS problem are generally applicable to
regular dependence algorithms.

We especially see potential for this technique at future
MIC systems where a large number of processing cores
are available. As these systems usually consist of
in-order architectures, especially our new variation of the su-
pernode transformation should be applied. The techniques
and concepts presented in this paper may be improved
by applying software pipelining. Also, how to port the
presented techniques to SIMD architectures such as most
recent GPGPU’s will be conducted in future work.

References

[1] 1003.1 standard for information technology portable operating
system interface (posix) rationale (informative). IEEE Std 1003.1-

than you think. In PACT ’13, pages 7–16, Juan-les-Pins, France,
September 2004. 4

dynamic programming algorithm for solving the longest common
subsequence problem. In Computer Systems and Applications
(ACSAC). 2010 IEEE/ACS International Conference on, pages 1–5,
May 2010. 2, 5

subsequence algorithms. In String Processing and Information
Symposium on, pages 39–48, 2000. 2

Shared Memory Parallel Programming (Scientific and Engineering
Computation). The MIT Press, 2007. 4


computer algorithm for the longest common subsequence problem.
In Parallel, Distributed and Network-Based Processing, 2003. Pro-
2003. 1, 2, 3

supernode shape. Parallel and Distributed Systems, IEEE Transac-


minimized total running time. Parallel and Distributed Systems,

and Distributed Systems, IEEE Transactions on, 13(12):1220 –
1233, dec 2002. 2, 4, 8

implementable systolic algorithm for the longest common sub-
sequence problem. In Machine Learning and Cybernetics, 2008 Inte-
national Conference on, volume 6, pages 3150 –3155, July
2008. 2

for the multiple longest common subsequence (mLcs) problem. In
Parallel Processing, 2008. ICPP ’08. 37th International Conference
on, pages 354 –363, sept. 2008. 2

N. Satish, M. Smelyanskiy, S. Chenmutpy, P. Hammar-
lund, R. Singhal, and P. Dubey. Debugking the 100x gpu vs. cpu
myth: an evaluation of throughput computing on cpu and gpu.
In Proceedings of the 37th annual international symposium on
Computer architecture, ISCA ’10, pages 451–460, New York, NY,
USA, 2010. ACM. 4, 7

algorithm. In Consumer Electronics, Communications and Networks
(CECNet), 2011 International Conference on, pages 4338 –4341,
April 2011. 1

[16] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey,
S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Gro-
chowski, T. Juan, and P. Hanrahan. Larraybe: a many-core x86
architecture for visual computing. In ACM SIGGRAPH 2008
papers. SIGGRAPH ’08, pages 18.1–18.15, New York, NY, USA,
2008. ACM. 4, 7

40(6):723–742, 1991. 3, 4