On Parallel Evaluation of Matrix-Based Dynamic Programming Algorithms

David Bednárek, Michal Brabec, and Martin Kruliš

Parallel Architectures/Algorithms/Applications Research Group
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
{bednarek,brabec,krulis}@ksi.mff.cuni.cz

Abstract. Dynamic programming techniques are well established and employed by various practical algorithms, for instance the edit-distance algorithm. These algorithms usually operate in iteration-based fashion where new values are computed from values of the previous iterations, thus they cannot be processed by simple data-parallel approaches. In this paper, we investigate possibilities of employing multicore CPUs and Xeon Phi devices for efficient evaluation of a class of dynamic programming algorithms. We propose a parallel implementation of Wagner-Fischer algorithm for Levenshtein edit-distance that demonstrates utilization of task parallelism and SIMD parallelism which are native to both architectures. The proposed solution has been subjected to empirical evaluation and its results are also presented in this work.

Keywords: dynamic programming, edit distance, parallel, SIMD, MIC

1 Introduction

Dynamic programming is a well established method of algorithm design. It solves complex problems by breaking them down into simpler subproblems. It is especially useful when the subproblems overlap and identical subproblems are computed only once, hence redundant computations are avoided. On the other hand, algorithms based on dynamic programming are usually difficult to parallelize, since the subproblems are interdependent – i.e., one subproblem requires the results of previous subproblems.

In this work, we focus on dynamic programming algorithms which logically organize their subproblem results into regular matrix. Each result in the matrix is computed from a small subset of previous results, using a recurrent formula like

$$x_{i,j} = f_{i,j}(x_{i-1,j}, x_{i-1,j-1}, x_{i,j-1})$$

Typical examples are the edit distance problem originally described by Levenshtein [5], the dynamic time warping algorithm [8], or the Smith-Waterman algorithm [11] for molecular sequence alignment.

The functions $f_{i,j}$ are often very simple, as it is in the case of the Wagner-Fischer dynamic programming algorithm [12] for the Levenshtein distance,

$$f_{i,j}(p,q,r) = min(p+1, q+\delta_{u[i]}^{v[j]}, r+1)$$

where the Kronecker δ compares the i-th and j-th positions in the input strings u and v, respectively.

For our implementation and experiments, we have selected the Levenshtein distance as a representative of the examined class of problems. We will not employ any optimizations designed specifically for the Levenshtein distance (like the Myers' algorithm [9]), so our proposed solution is applicable for similar dynamic programming problems as well.

The dependencies between individual invocations of the recurrent formula f significantly limit the parallelism available in the problem – for a $M \times N$ matrix, at most $min(M,N)$ elements may be computed in parallel, using the diagonal approach illustrated in Figure 1. In addition, the computation of f is typically too small unit of work to become a base of thread-level parallelism; therefore, the pure diagonal approach is not applicable in real hardware.

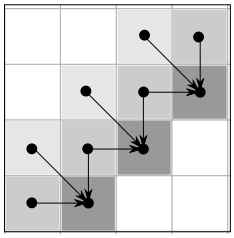

Fig. 1. Dependencies in the matrix and the diagonals

In this paper, we examine parallel implementation of this class of problems in two hardware environments – multicore CPUs and the new Xeon Phi devices which are based on the Intel Many Integrated Core (MIC) architecture [4]. Both the multicore CPUs and Xeon Phi devices employ two levels of parallelism: The multicore nature of both devices allows concurrent execution of multiple independent threads and each core is capable of executing vector (SIMD) instructions.

We present an implementation of the Levenshtein distance problem which employs both thread-based and SIMD-based parallelism. We compare their performance in multicore and manycore architectures and we show that both levels of parallelism offer very significant speedup, allowing a manycore chip to execute about $6 \cdot 10^{10}$ invocations of the recurrent formula per second.

The paper is organized as follows. Section 2 overviews work related to efficient implementations of dynamic programming algorithms. In Section 3, we revise the fundamentals of multicore CPUs and Xeon Phi devices which influenced our implementation decisions. Section 4 briefly analyses the problem with respect to parallelism and cache hierarchy, while the proposed algorithm is described in

Section 5. The empirical evaluation is summarized in Section 6 and Section 7 concludes the paper.

2 Related Work

One of the first papers that addressed parallel evaluation of edit distance problem was presented by Mathies [7]. It was a theoretical work that described and algorithm for PRAM execution model. The proposed algorithm achieved $O(\log m \log n)$ time complexity using mn processors, where m and n are the lengths of the compared strings.

Most practical parallel algorithms are based on an observation of Delgado [1], who studied the data dependencies in the dynamic programming matrix. Two possible ways of processing the matrix were defined in their work – *unidirectional* and *bidirectional* filling. Their idea allows limited concurrent processing of independent parts, but it needs to be modified for massively parallel environment.

A similar problem that uses dynamic programming is *dynamic time warping* (DTW) as defined by Müller [8]. Weste et al. [13] proposed a parallel approach that implemented the algorithm directly in CMOS chip. Even though this solution is rather specialized, it proposed some basic ideas that may be applied in highly parallel environments. More recent discussion on parallelization techniques for GPUs and FPGAs was presented by Sart et al. [10]. Their work focused mainly on a specific version of DTW algorithm, which reduces the dependencies of the dynamic programming matrix, thus allows more efficient parallelization.

Another example of a problem suitable for dynamic programming is the Smith-Waterman algorithm [11], which is used for protein sequences alignment. Farrar presented a SIMD implementation [3], which utilizes instructions of mainstream CPUs. It achieved a 2-8× speedup over other SIMD implementations. There are also many papers that address the parallelization of Smith-Waterman on GPUs. Perhaps the most recent work was presented by Liu et al. [6] and it combines many observations from the previous work on the subject.

Unlike these related papers, our work does not focus on a single dynamic programming problems. We are proposing a parallel approach, which may be used for various dynamic programming problems sharing the matrix shape of dependencies.

3 Multicore and Manycore Architectures

The pursuit for higher CPU frequency has reached an impasse and it has been abandoned for concurrent processing. Parallelism has easily found its way into mainstream processors and current architectures employ parallel processing on several levels. Furthermore, massively parallel platforms have emerged, such as GPGPUs or new Xeon Phi devices.

Current state-of-the-art CPUs employ parallelism on several levels. The most obvious one (and perhaps the most advertised one) is the explicit task parallelism

which is realized by integrating multiple CPU cores into a single CPU chip. It is further magnified by multithreading technologies such as hyperthreading (Intel) or dual-core module (AMD), which equip each physical core with two (or even more) logical frontends that share the internal resources of the core but appear as independent cores to the rest of the system.

Another type of parallelism is embedded in every core on instruction level where independent subsequent instructions may overlap, since they are being processed by different circuits of the CPU core. In most cases, exploiting this type of parallelism is limited to code optimizations by a compiler.

Finally, the CPUs of the day also employ SIMD parallelism. It is implemented using specialized vector registers and instructions sets that operate on these registers, such as Intel SSE or AVX instructions. The vector registers are capable of comprising multiple values (e.g., in case of SSE, four 32-bit numbers) and the instructions perform their operations concurrently on all the values in the register. The SIMD instructions can be generated automatically by the compiler; however, in complicated algorithms, the utilization of SIMD instructions has to be designed by the programmer.

3.1 Xeon Phi Device

The Intel Many Integrated Core (MIC) architecture [4] takes the parallel processing one step further, both on the level of task parallelism and the SIMD parallelism. It is Intel's answer to rapidly evolving GPUs, which are currently excelling in data parallel tasks. This architecture was employed in the Xeon Phi devices, which are parallel accelerators designed as extension cards to servers or PCs (in a similar way as GPUs). The device comprises a massively parallel chip and several gigabytes of on-board memory.

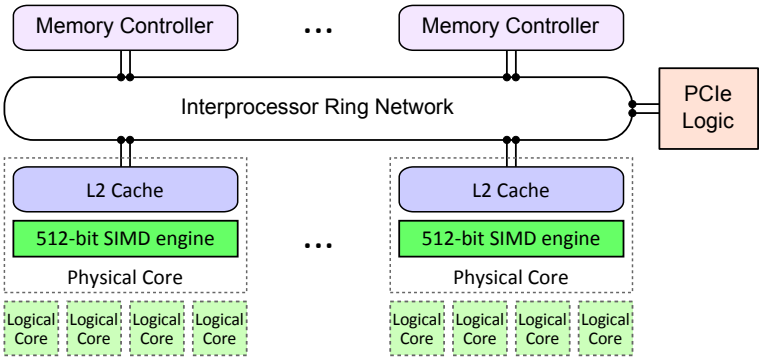

Fig. 2. An illustrative schema Xeon Phi processor

The Xeon Phi processor is based on older Pentium architecture, but is has been augmented in several ways. The most important differences from mainstream CPUs are the following:

- Simplified architecture allowed an order of magnitude increase in the number of integrated cores. The Xeon Phi has approximately 60 cores (exact figure depends on the model number).
- The cores employ in-order instruction execution which slightly limits the instruction parallelism. On the other hand, the chip implements four-way multithreading where each core is prefaced with four logical cores. Instructions from the four associated threads are interleaved on the physical core.
- The MIC architecture utilize vector instruction set for SIMD computing, where the registers are 512-bits wide – i.e., 4× wider than the SSE registers.
- The Xeon Phi chip has up to 8 dual-channel memory controllers, so it offers much higher memory throughput than common SMP or NUMA systems.

Let us summarize the observations made about CPUs and Xeon Phi devices and point out the most important implications for our code design:

The dynamic algorithm should be divided into concurrent tasks, so it can occupy tens (or better hundreds) computational cores. The vector instructions are very important, since SIMD execution units are present in both target architectures. Furthermore, algorithm design should be prepared for different sizes of SIMD registers.

4 Analysis

Each problem instance is characterized by a set T of its *elementary tasks*. In our case, T corresponds to a subset of the two-dimensional space $\mathbf{N} \times \mathbf{N}$ usually bounded by a rectangle. For Levenshtein distance, the dimensions of the rectangle corresponds to the sizes of the input strings. Each elementary task $T_{i,j}$ corresponds to computing $x_{i,j}$ using the function $f_{i,j}$.

A parallel algorithm may divide T into a hierarchy of *subtasks*. A set $S \subseteq T$ is a correct subtask when, for every pair of dependent elementary tasks in S, all dependency paths between them are also contained in S.

Each subtask has its i- and j-*projections*: $P_i(S) = \{i \mid \langle i,j\rangle \in S\}$, $P_j(S) = \{j \mid \langle i,j\rangle \in S\}$. These projections help to estimate the size of subtask inputs and outputs:

The input $I(S)$ of a subtask consists of $x_{i,j}$ values outside S which are input to some elementary tasks in S. Similarly, the output $O(S)$ consists of $x_{i,j}$ values inside S which are read by some elementary tasks outside S. A simple geometric observation yields the following limits for their sizes:

$$|I(S)| \geq |P_i(S)| + |P_j(S)| + 1 \qquad |O(S)| \geq |P_i(S)| + |P_j(S)| - 1$$

If the projections $P_i(S)$ and $P_j(S)$ are contiguous (i.e., intervals in $\mathbf{N}$), then the constraints become equalities.

At the same time, the size of a subtask is trivially constrained by

$$|S| \leq |P_i(S)| \cdot |P_j(S)|$$

Consequently, the following relation holds:

$$\rho(S) = \frac{|I(S)| + |O(S)|}{|S|} \geq 2 \cdot \frac{|P_i(S)| + |P_j(S)|}{|P_i(S)| \cdot |P_j(S)|}$$

The ratio $\rho(S)$ denotes how many inputs and outputs must be transferred per a unit of work, provided the communication inside the subtask is done using a local memory. The ratio is important in determining cache efficiency - lower $\rho(S)$ means that distributing data between subtasks requires less transfers.

This observation implies that the best cache efficiency will be achieved for square subtasks. For subtasks of the same shape, the $\rho(S)$ ratio is proportional to $|S|^{-1/2}$. It means that the shape is important for small subtasks which are likely to suffer from limited communication bandwidth. In large subtasks, the outer communication will likely become negligible compared to their inner complexity.

The projections also give constraints on parallel execution: Subtasks $S_1, \ldots, S_n$ may run in parallel only if they are independent, which implies that their projections must be disjoint in both dimensions. Consequently, the size of projections is limited by

$$\sum_{k=1}^{n} |P_i(S_k)| \leq |P_i(T)|$$

where T is the supertask divided into $S_1, \ldots, S_n$ (analogically the j-projection). Since load balancing requires subtasks of similar sizes, $|P_i(S_k)|$ is limited by $|P_i(T)|/n$ and, consequently,

$$|S_k| \leq \frac{|P_i(T)| \cdot |P_j(T)|}{n^2}$$

This constraint immediately explains the inherent difficulty in parallelization of our dynamic programming problems: The maximal available size of the subtasks is inversely proportional to the square of the required degree of parallelism.

Furthermore, the communication ratio $\rho(S_k)$ is proportional to $|S_k|^{-1/2}$ and thus to n. Consequently, higher degree of parallelism induces higher communication costs per unit of work.

5 Algorithms

The analysis in the previous section offers a guidance on the construction of parallel algorithm for our problem: The complete task must be divided into a sufficient number of subtasks, using subtasks of approximately square shape. The subtasks will form a two-dimensional mesh, where only those residing on a diagonal may be executed in parallel.

Inside every subtask, a different form of parallelism is possible using vector instructions. However, the degree of vector parallelism is usually small (4 to 16 in our environment). At the same time, it is advantageous to maintain the lowest level of subtasks so small that they can handle internal communication using

CPU registers. To fit in the limited space, one of the subtask projections must be sufficiently small, typically equal to the size of a vector register.

In our algorithm, such subtasks are called *stripes*. Each stripe is s elements wide and $s \cdot n$ elements long, where s is the size of a vector register and n is a carefully tuned constant. To allow vector-based parallelism, stripes are not rectangular but skewed as shown in Fig. 3.

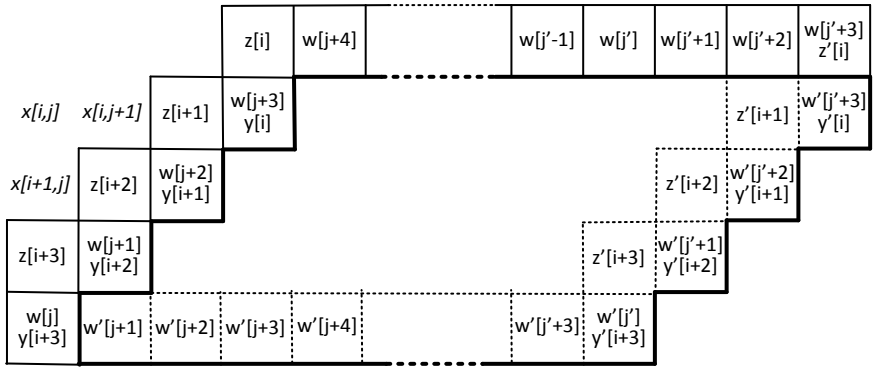

Fig. 3. Stripe

Figure 3 also illustrates the meaning of *boundary arrays* used in our algorithm to replace the two-dimensional array $x[1:N, 1:M]$. The $w[1:M]$ array is stretched along the (horizontal) j-dimension and each $w[j]$ stores the value of $x[i,j]$ for the greatest (lowermost) i for which the task $T_{i,j}$ was already completed. Similarly, $y[i] = x[i,j]$ and $z[i] = x[i-1,j]$ for the greatest (rightmost) j for which $T_{i,j}$ is completed. The figure shows the boundary arrays before the stripe execution while the primed versions w', y', z' denote the meaning after the stripe execution. The $x[i,j]$ mark denotes the reference point used to specify the position of the stripe in Algorithm 1 and 2.

For smaller n, a stripe is too small to form a base for thread-level parallelization; selecting larger n would violate the requirement for approximately square shape. Therefore, n stripes are grouped (by sequential execution) into a larger subtask of size $s \cdot n$ times $s \cdot n$. Because of these dimensions, we call them *square subtasks*, although they are skewed to the shape of parallelogram.

The constant n is tuned to achieve best performance: Selecting greater n diminishes the overhead of parallel task creation and it also improves the communication ratio $\rho(S)$. On the other hand, larger subtasks lead to worse load balancing. In addition, a subtask whose internal communication can fit inside the L1 cache runs faster; therefore, the size $s \cdot n$ shall be sufficiently small, proportionally to the cache size. In our environment described in Sec. 6, selecting $s \cdot n = 256$ produces best or near-the-best results in both multicore and manycore cases.

The square subtasks form the base of parallel evaluation. A diagonally shaped set of square subtasks is selected for simultaneous execution; when they are completed, subsequent diagonal may be launched. Thus, the complete algorithm consists of a sequential loop over *diagonals* as shown in Algorithm 1. Due to the skewed shape of stripes, the algorithm must deal with incomplete stripes at the boundaries of the problem rectangle – the incomplete stripes are evaluated using scalar operations.

Algorithm 1 Iteration over stripes

Require: N, M – problem size, s – SIMD vector size, n – stripe length
  $u[1:N], v[1:M]$ – input sequences
  $y[1:N], z[1:N], w[1:M]$ – initial boundary values
Ensure: $y[1:N], z[1:N], w[1:M]$ – final boundary values
  for $p := 1$ to $\lceil (M+N)/(n \cdot s) \rceil$ **do**
    for $q := 1$ to $\lceil N/(n \cdot s) \rceil$ **in parallel do**
      for $r := 1$ to $\lceil N/s \rceil$ **do**
        $i := n \cdot s \cdot q + s \cdot r - (n+1) \cdot s + 1$
        $j := n \cdot s \cdot p - 2 \cdot n \cdot s \cdot q - s \cdot r + n \cdot s$
        if stripe $\langle i, j \rangle$ intersects with the rectangle $\{1:N\} \times \{1:M\}$ **then**
          if stripe $\langle i, j \rangle$ is a subset of the rectangle $\{1:N\} \times \{1:M\}$ **then**
            evaluate stripe $\langle i, j \rangle$ using SIMD
          else
            evaluate stripe $\langle i, j \rangle$ using scalar operations
          end if
        end if
      end for
    end for
  end for

Stripe evaluation consists of updating boundary values stored in the arrays $y[1:N], z[1:N], w[1:M]$, at the indexes corresponding to the i- and j-projections of the stripe. Within a diagonal, the square subtasks have disjoint projections; therefore, their memory accesses do not overlap. Nevertheless, alignment to cache-line boundary must be observed to avoid false sharing – therefore, n must be a multiple of cache-line size divided by the SIMD vector size.

Algorithm 2 performs SIMD evaluation of a stripe. It calls a vectorized version $\overline{f}$ of the elementary task function f, which evaluates s elementary tasks at once. In addition, the algorithm requires several vector operations to manage the vector buffers. The operations are taken from the Intel® SSSE 3 instruction set and their semantics is shown in Table 1. The upper half of the table summarizes the operations required by the generic part of the algorithm, the lower half contains operations required to implement the specific Levenshtein distance problem as shown in the following definition:

$$\overline{f}(\overline{w}, \overline{z}, \overline{y}, \overline{u}, \overline{v}) = \mathsf{add}(\mathsf{min}(\overline{w}, \mathsf{add}(\overline{z}, \mathsf{cmpeq}(\overline{u}, \overline{v})), \overline{y}), \mathsf{broadcast}(1))$$

Algorithm 2 Vectorized stripe algorithm

Require: $\langle i, j \rangle$ – stripe position, s – vector size, n – stripe length in units of s
$u[i : i + s - 1]$, $v[j : j + (n + 1) \cdot s - 1]$ – input sequences
$y[i : i + s - 1]$, $z[i : i + s - 1]$, $w[j : j + (n + 1) \cdot s - 1]$ – status vectors
Ensure: $y[i : i + s - 1]$, $z[i : i + s - 1]$, $w[j : j + (n + 1) \cdot s - 1]$ – updated status vectors
$v' := \mathsf{reverse}(v[j : j + s - 1])$
$u' := u[i : i + s - 1]$; $y' := y[i : i + s - 1]$; $z' := z[i : i + s - 1]$
for $k := 1$ to n **do**
  $v'' := \mathsf{reverse}(v[j + k \cdot s : j + (k + 1) \cdot s - 1])$
  $w' := \mathsf{reverse}(w[j + k \cdot s : j + (k + 1) \cdot s - 1])$
  for $m := 1$ to s **do**
    $v' := \mathsf{alignr}_1(v'', v')$; $z'' := \mathsf{alignr}_1(w', y')$
    $y'' := \overline{f}(z'', z', y', u', v'))$
    $v'' := \mathsf{shiftr}_1(v'')$; $w' := \mathsf{alignr}_1(y', w')$
    $y' := y''$; $z' := z''$
  end for
  $w[j + (k - 1) \cdot s : j + k \cdot s - 1] := \mathsf{reverse}(w')$
end for
$w[j + k \cdot s : j + (k + 1) \cdot s - 1] := \mathsf{reverse}(y')$
$y[i : i + s - 1] := y'$; $z[i : i + s - 1] := z'$

The algorithm uses vector variables $u', v', v'', y', y'', z', z'', w', v''$ to cache a portion of the input strings u, v and boundary arrays y, z, w. Some of these variables are shifted using alignr or shiftr instructions – thus, any misaligned reads or writes of the arrays are avoided.

The $\mathsf{reverse}$ operations are required to properly align the order of elements of arrays u, y, z with the arrays v, w, because the $\overline{f}$ function must act on independent elements on an (anti-)diagonal. Although $\mathsf{reverse}$ may be implemented using a vector instruction, it may be eliminated from the algorithm by reversing the order of elements in the v, w arrays before invocation of the algorithm.

$\mathsf{reverse}(\langle x_1, x_2, \ldots, x_s \rangle)$	$= \langle x_s, x_{s-1}, \ldots, x_1 \rangle$
$\mathsf{alignr}_1(\langle x_1, \ldots, x_s \rangle, \langle y_1, \ldots, y_s \rangle)$	$= \langle x_s, y_1, \ldots, y_{s-1} \rangle$
$\mathsf{shiftr}_1(\langle x_1, \ldots, x_s \rangle)$	$= \langle 0, x_1, \ldots, x_{s-1} \rangle$
$\mathsf{broadcast}(x)$	$= \langle x, \ldots, x \rangle$
$\mathsf{cmpeq}(\langle x_1, \ldots, x_s \rangle, \langle y_1, \ldots, y_s \rangle)$	$= \langle -\delta_{x_1}^{y_1}, \ldots, -\delta_{x_s}^{y_s} \rangle$
$\mathsf{min}(\langle x_1, \ldots, x_s \rangle, \langle y_1, \ldots, y_s \rangle)$	$= \langle min(x_1, y_1), \ldots, min(x_s, y_s) \rangle$
$\mathsf{add}(\langle x_1, \ldots, x_s \rangle, \langle y_1, \ldots, y_s \rangle)$	$= \langle x_1 + y_1, \ldots, x_s + y_s \rangle$

Table 1. Vector operations used in the SIMD algorithm

6 Experiments

The experiments were conducted on various string lengths, with 32-bit characters. We present only the results for strings of equal length, since they produce a square matrix that provides the greatest opportunity (and challenge) for parallel computation. Furthermore, we present results for a limited range of string lengths. Significantly shorter strings limit the parallelism, since the corresponding dynamic programming matrix does not have sufficient size. Larger datasets exhibit similar behavior in the terms of relative speedup as the presented ones.

The algorithms were implemented in C++11, using Intel SSE intrinsic functions for SIMD instructions and Intel Threading Building Blocks for task parallelism.

The experiments were conducted on a server equipped with four Intel Xeon E5-2630 CPUs (total of 24 cores) and an Intel Xeon Phi accelerator card with 61 cores, running Red Hat Enterprise Linux 7 and using Intel C++ Compilers for both CPU and Xeon Phi versions.

We measured wall time using standard C++11 clock interface whose precision was satisfactory given the run times were between seconds and thousands of seconds. All performed tests were repeated five times and the related measurements were within 1% range. We present the average value of these five measured values as the result of each test.

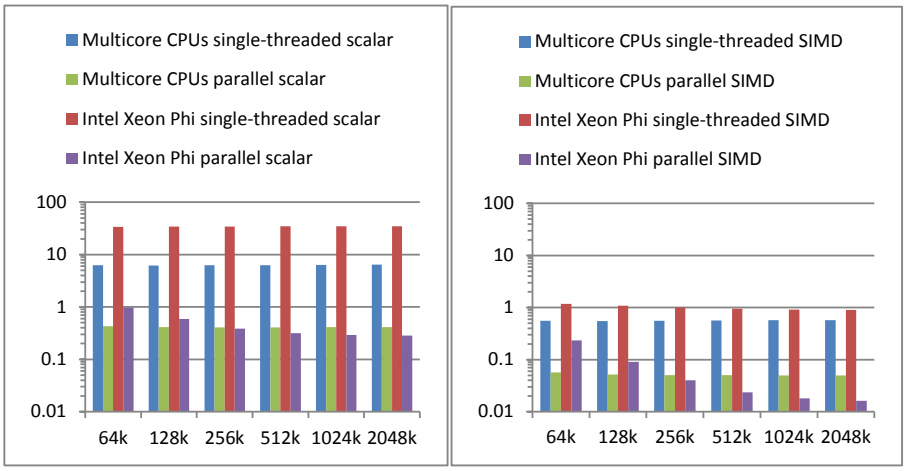

Fig. 4. Execution time in nanoseconds per elementary task

Figure 4 shows the execution time of various Xeon Phi and CPU implementations. The horizontal axis contains six different string sizes between 64000 and 2048000 elements. On the logarithmical vertical axis, the wall time is displayed divided by the product of string lengths. In other words, the vertical axis displays the time in nanoseconds per elementary task, where an elementary task consists of the computation of one 32-bit element in the matrix.

The left graph shows the results for scalar implementation while the right graph displays SIMD versions – four-fold vectorization (SSE) in the case of CPU and 16-element vectors (AVX-512) in the case of Intel Xeon Phi. In each graph, results for multicore CPU and Intel Xeon Phi are compared. In addition to the parallel version based on Intel TBB, single-threaded versions were also measured.

Each of the eight versions was independently manually tuned to achieve the best performance for strings of 100000 elements. Afterwards, the measurements for the six different string sizes were made with the same configuration, i.e., the graph shows scaling without any retuning.

For all except the parallel Xeon Phi versions, the graphs show very little dependence on the string size. Since the graphs display the time per elementary task, this fact demonstrates that, in the given range of sizes, the total time is almost precisely proportional to the product of string sizes. However, for parallel Xeon Phi, the time per elementary task significantly decreases until the string size reaches approximately one million. This shows that, for smaller strings, the computing power available in the 61 cores of Xeon Phi is not efficiently employed. Note that the implementation was tuned on data of similarly small size (100k) and the tuning included the selection of task size; thus, the inferior performance means that there is still not enough parallelism available in our algorithm for smaller data sizes. Nevertheless, there might be other factors that contributed to the performance degradation, in particular, the fact that the task-stealing strategy of Intel TBB is unaware of the complex cache structure of Xeon Phi.

Both the CPU and Xeon Phi versions show significant speed gain when vectorized (compare the two graphs in Fig. 4). The ratio is even greater than the degree of vectorization: The four-fold vectorization of the parallel CPU version brings about eight-fold speedup; similarly, the SIMD-based speedup in Xeon Phi is almost 18 in parallel version and almost 40 in single-threaded version, although the AVX-512 offers only 16 32-bit operations at once. This counterintuitive observation has an explanation: The algorithm is designed in cache-aware way, including the use of registers as L0-cache. Since the vector registers offer greater total space than the scalar registers, the vectorized version requires less bytes transferred between the registers and the L1-cache. At the same time, the use of vector reads/writes offers multiplied throughput; together, the total speed-up may be greater than the vector size.

7 Conclusions

We have proposed a scalable approach for implementing a class of dynamic programming algorithms. It utilizes both task-parallelism and SIMD parallelism, which are strongly supported in current CPUs and Xeon Phi devices. The experimental evaluation suggests that the solution scales well if the input data are sufficiently large. Furthermore, the results confirm applicability of the many-core architecture in this class of problems, since a single Xeon Phi outperformed four six-core CPUs by the factor of three.

The results also revealed that the use of SIMD instructions is crucial for the performance of the Xeon Phi, offering speed-up factor greater than the intuitively expected 16 due to the effect of total register size.

Some questions are still left open; in particular, the contribution of the task-stealing strategy to the inferior performance for smaller strings at Intel Xeon Phi. This could be an opportunity to apply a NUMA-aware task scheduling strategy which we previously developed [2].

Acknowledgements

This paper was supported by Czech Science Foundation (GAČR), projects P103-14-14292P and P103-13-08195, and by the Charles University Grant Agency (GAUK) project 122214.

References

1. Delgado, G., Aporntewan, C.: Data dependency reduction in dynamic programming matrix. In: Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on. pp. 234–236. IEEE (2011)
2. Falt, Z., Kruliš, M., Bednárek, D., Yaghob, J., Zavoral, F.: Locality aware task scheduling in parallel data stream processing. In: Intelligent Distributed Computing VIII, pp. 331–342. Springer (2015)
3. Farrar, M.: Striped Smith–Waterman speeds database searches six times over other SIMD implementations. Bioinformatics 23(2), 156–161 (2007)
4. Intel: Xeon Phi Coprocessor `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`
5. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. In: Soviet physics doklady. vol. 10, p. 707 (1966)
6. Liu, Y., Wirawan, A., Schmidt, B.: Cudasw++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. BMC bioinformatics 14(1), 117 (2013)
7. Mathies, T.R.: A fast parallel algorithm to determine edit distance (1988)
8. Müller, M.: Dynamic time warping. Information retrieval for music and motion pp. 69–84 (2007)
9. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM (JACM) 46(3), 395–415 (1999)
10. Sart, D., Mueen, A., Najjar, W., Keogh, E., Niennattrakul, V.: Accelerating dynamic time warping subsequence search with GPUs and FPGAs. In: Data Mining (ICDM), 2010 IEEE 10th International Conference on. pp. 1001–1006. IEEE (2010)
11. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of molecular biology 147(1), 195–197 (1981)
12. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM (JACM) 21(1), 168–173 (1974)
13. Weste, N., Burr, D.J., Ackland, B.D.: Dynamic time warp pattern matching using an integrated multiprocessing array. IEEE transactions on computers 32(8), 731–744 (1983)