RESEARCH ARTICLE

Efficient Parallel Solutions to the Integral Knapsack Problem on Current Chip-multiprocessor Systems

Hammad Rashid\textsuperscript{a} and Clara Novoa\textsuperscript{b} and Mark McKenney\textsuperscript{a} and Apan Qasem\textsuperscript{a,}\textsuperscript{*}

\textsuperscript{a}Computer Science Department, Texas State University, San Marcos, TX, USA; \textsuperscript{b}Ingram School of Engineering, Texas State University, San Marcos, TX, USA

\textit{(v1.0 released June 2010)}

The emergence of chip multiprocessor systems has dramatically increased the performance potential of computer systems. However, harnessing the full potential of these systems depends largely on the effectiveness of system software such as compilers, in exploiting the on-chip parallelism. Additionally, since the amount of parallelism extracted by a compiler is directly influenced by the selection of the algorithm, algorithmic choice also plays a critical role in achieving a high fraction of peak performance. Hence, in the era of multicore computing, it is imperative that we reevaluate and rethink algorithms for key problem domains. This paper investigates the impact of algorithmic choice on the performance of parallel implementations of the integral knapsack problem on multicore architectures. The study considers two classes of algorithms and several algorithmic variants and evaluates each implementation based on a variety of performance metrics including data locality and sharing, granularity of parallelism, and scalability. The paper presents experimental results that show how each performance factor is affected by the selection of algorithm, changes in the input data set and variations in architectural characteristics such as cache capacity and degree of cache sharing.

\textbf{Keywords:} parallel algorithms, optimization problems, multicore, data locality, compilers

1. Introduction

It is widely agreed, that the industry trend of packing more and more cores on a single chip, brought on by the advent of multicore technology, is likely to continue for the next several years - perhaps decades. This fundamental shift in processor design technology implies that software will have to play a key role in harnessing the true potential of any computer system. In particular, compilers need to uncover parallelism at different levels and transform code for parallel execution. Also, runtime systems need to schedule parallel threads for efficient utilization of underlying architectural resources. For many problem domains, however, advances in performance optimizing software will not be sufficient. To a great extent, the amount of parallelism that can be extracted by the compiler is determined by the initial choice of the algorithm. For example, both dynamic programming and branch-and-bound algorithms are used to solve combinatorial optimization problems, however, these algorithms exhibit different degrees of parallelism and parallel implementations based on these two algorithms can have widely varying performance. Thus, it
is important to consider the issue of algorithmic choice when implementing parallel solutions on current chip multiprocessor (CMP) architectures.

For many problems, several efficient solutions exist. However, which algorithm will perform best on any given situation depends on a host of factors including the size and shape of the data set, the optimizations performed by the compiler and architectural parameters such as the size of the cache and depth of the instruction pipeline. Finding the most suitable algorithm that will deliver high-performance across different architectures, for different problem sizes and shapes, has always been a significant challenge and many researchers have addressed this issue over the years. In particular, approaches based on automatic tuning have been quite successful for automating the selection of the optimal (or near optimal) algorithmic variant for specific domains [5, 7, 23]. However, the emergence of chip multiprocessor systems adds a new dimension to the problem making it more challenging. This new dimension relates to the fact that in CMP architectures there is one or more levels of cache, shared among multiple processing cores. Shared cache poses an inherent trade-off between data locality and parallelism [21]. On one hand, any parallel decomposition of the application will inevitably influence the data access patterns for each concurrent thread. On the other hand, any transformations for improving locality will impose constraints on parallelism. For example, if an application is parallelized without considering locality it will affect performance due to reduced locality, whereas, if all attention is devoted to improving locality, the performance will be affected due to unexploited parallelism. Thus, when parallelizing an application for multicore architectures, it is imperative that we find the right balance between data locality and parallelism. Since the choice of an algorithm dictates the amount of exploited parallelism, and in turn exploited data locality in the program, algorithmic choice is certain to play a major role in obtaining high-performance on chip multiprocessor systems.

This paper studies the impact of algorithmic choice on the performance of parallel algorithms for the integral knapsack problem (IKP) under a dynamic programming (DP) approach. IKP is an important problem in the domain of combinatorial optimization because it directly models many practical situations such as capital budgeting, cutting stock [9, 10] and cargo loading problems [4]. Furthermore, IKP’s appear as sub-problems in more complex situations such as the multi-dimensional cutting stock problem [11] and some other set-partitioning problems. Integral and 0/1 knapsack problems have also contributed to developing solution methods for general integer programming problems (IP), specifically in the generation of minimal cover induced constraints and in coefficient reduction procedures for strengthening bounds [15]. The relationship between KP’s and other IP’s has motivated great interest for developing efficient solutions for KP’s. As a consequence, KP’s are fairly well studied problems, although there is less work on integral KP’s than on 0/1 KP’s. Several factors make IKP an interesting candidate for this study. First, IKP algorithms under a DP approach make themselves amenable to different types of parallelism. In particular, there are algorithms that can be parallelized in a pipelined fashion [3] and there are algorithms, where the central loop lends itself to a data parallel decomposition. Second, most implementations of IKP exhibit data locality that can be exploited through compiler transformations. However, as mentioned above, the issue of data locality gives rise to interesting trade-offs for parallel applications on multicore architectures. The third factor that motivates this research to focus on IKP, is that the performance of different algorithm variants is sensitive to the size and shape of the input data sets. Thus, IKP is a suitable target for evaluating the impact of algorithmic choice.

The primary goal of this performance study is not to find the optimal paral-
level algorithm for IKP on multicore architectures but rather to understand the various performance trade-offs from choosing a particular type of algorithm. To pinpoint performance issues and identify bottlenecks, we analyze each algorithmic variant and utilize hardware performance counters to measure a variety of metrics at thread-level granularity. The study looks at several performance issues common to any parallel implementation on multicore architectures including data sharing and affinity, synchronization, and intra-node thread scheduling. More specifically, this paper makes the following contributions:

- We present a quantitative analysis of various performance aspects for different algorithmic variants of the IKP. To the best of our knowledge, no previous work has looked at performance issues related to IKP on multicore architectures
- We identify a tunable parameter for several algorithmic variants, that can be used to significantly enhance performance
- We evaluate the impact of dominance-based optimizations on the execution time of several algorithmic variants
- We analyze the impact of data-set characteristics on execution time

The rest of the paper is organized as follows: in Section 2 we provide background on the integer knapsack problem; Section 3 provides results from an experimental study conducted on sequential versions of each algorithm; Section 4 describes our strategy for parallelizing knapsack algorithms; Section 5 presents the main experimental results; Section 6 presents related work and finally, we conclude in Section 7.

2. Background

2.1 The Integral Knapsack Problem

IKP can be formulated as follows: Given a knapsack of capacity \( C \) and a set of \( n \) different objects (items), each one of them with profit \( p_j \) and weight \( w_j \), find non-negative integers \( x_1, \ldots, x_n \), where \( x_j \) represents the number of \( j^{th} \) type objects, such that the total weight of the objects does not exceed the knapsack capacity and the total profit is maximized. In the IKP, \( w_j, p_j, n, \) and \( C \) are all positive integers. If \( x_i \in 0, 1 \) the problem reduces to the 0/1 knapsack problem.

Some authors refer to the IKP as unbounded knapsack problem (UKP) [15]. The UKP assumes that an infinite number of objects of each kind are available while the bounded knapsack problem (BKP) assumes that there is up to \( b_j \) objects of each type available, that is, \( x_j = 1, \ldots, b_j \). Following is the IP formulation for the UKP:

Maximize

\[
\sum_{j=1}^{n} p_j x_j
\]

subject to

\[
\sum_{j=1}^{n} w_j x_j \leq C \quad j = 1, \ldots, n
\]

\[
x_j \geq 0 \text{ and integer}, \quad j = 1, \ldots, n
\]
Even if knapsack problems (KP’s) look as the simplest IP’s, they are NP-hard [8]. Thus, KP’s cannot be solved in a time bounded by a polynomial in $n$ [15]. However, they can be solved with pseudo-polynomial algorithms since $\log_2(C)$ bits are required to encode the input $C$.

2.2 Solution approaches

The classic approaches for solving exact KP’s are branch and bound (B&B) [15] and dynamic programming (DP) [2, 3, 14]. Here, the discussion focuses on DP. Hybrid approaches combining DP and IP are also mentioned briefly.

2.2.1 DP forward recursion 1

Works in [9] and [15] presented a DP forward recursion (see Eq. 1) to compute $f_m(\hat{c})$, the total profit (i.e. total value) from loading the most valuable combination if considering $m$ items and a knapsack capacity $\hat{c}$. In this recursion, $l$ represents a possible number of items to load.

$$f_m(\hat{c}) = \max\{f_{m-1}(\hat{c} - lw_m) + lp_m\}$$

$l$ integer, $0 \leq l \leq \lfloor \hat{c}/w_m \rfloor$ (1)

where $m = 1, 2, ..., n$ and $\hat{c} = 0, ..., C$. After the forward recursion step, the optimal value for the profit function is given by $f_n(C)$. A backtracking step permits to determine the optimal solution $(l^*_m, m = 1, 2, ..., n)$ associated to $f_n(C)$.

For each $m$, $O(C\lfloor \hat{c}/w_m \rfloor)$ operations are required to find $f_m(\hat{c})$ and then the overall time complexity is $O(C \sum_{m=1}^n \lfloor \hat{c}/w_m \rfloor)$ or $O(nC^2)$ in the worst case. The space complexity is $O(nC)$, since for each item, the vector $f_m(\hat{c})$ must be stored.

2.2.2 DP forward recursion 2

The work in [6] presented a recursion to the 0/1 knapsack problem which was extended to the integral case by [11] and parallelized by [16]. Authors in [11] mention that this recursion (see Eq. 2) is more efficient than the one in Eq. 1. The recursion is given by:

$$f_m(\hat{c}) = \max\{p_m + f_m(\hat{c} - w_m), f_{m-1}(\hat{c})\}$$

$m = 1, 2, ..., n$ and $\hat{c} = 0, ..., C$. Equation 2 selects between loading or not a unit of product $m$. This recursion involves less operations than the one in equation 1 by a factor of $1/n \sum_{m=1}^n \lfloor (\hat{c}/w_m) \rfloor$ and the resulting time and space complexity are $O(nC)$. Details about the procedures to find the optimal solution $(l^*_m, m = 1, 2, ..., n)$ associated to $f_n(C)$ are in [11] and [12].

For large IKP’s the B&B approach seems more efficient than DP [3]. The instances that B&B can solve are usually larger than the ones DP can solve. However, in contrast to B&B algorithms, DP recursions (equations 1 and 2) are insensitive to the parameters $p_i, w_i$, $i = 1, 2, ..., m$ and therefore they can be good for non well-behaved problems (i.e. problems with correlated $p_i$ and $w_i$ values) but not so good for well-behaved problems (i.e. uncorrelated problems). DP also exhibits two additional advantages: knowledge of the solution for any capacity lower than the given maximal capacity and ability to reuse the known solutions for solving larger capacity problems [2].

2.2.3 Hybrid approaches

The work in [18] is the first hybrid approach for solving IKP’s. The algorithm significantly outperforms all existing algorithms for solving the problem. The algo-
The work in [10] introduces the concept of dominance to reduce the size of the search space (number of states) in the DP approach. Simple dominance (sd) states that if an object type has a larger weight and smaller profit than another, the former may never occur in an optimal solution. Thus, object \( i \) is simply dominated by object \( j \) when \( \text{w}_i > \text{w}_j \) and \( \text{p}_i < \text{p}_j \).

In [15] multiple dominance (md) is introduced. It states that object \( i \) is multiple dominated by object \( j \) if and only if \( \lceil \text{w}_i / \text{w}_j \rceil \geq \text{p}_i / \text{p}_j \). This dominance relation means that \( j \) dominates \( i \) when profit from all objects of type \( j \) that can be allocated in the space occupied by object \( i \), is larger than the profit for \( i \). Refined dominance relationships such as the collective dominance and threshold dominance have been proposed and used in [2].

3. DP-based Algorithms for the Integral Knapsack Problem

This research investigates algorithms based on DP forward recursion 1 and 2, as described in Section 2.2. In the rest of the paper, these two algorithms are referred to as classic and morales, respectively. Both algorithms use a two dimensional matrix \( M \) of \( n \) rows that represent the items and \( C \) columns that represent the knapsack capacities. The indices \((m, \hat{c})\) is used to represent any \((\text{row, column})\) pair in the grid. The knapsack capacity is in terms of weight but it can be used for any other relevant problem dimension, such as volume, length, etc. The goal of each algorithm is to compute the maximum attainable profit (value) from selecting any combination of items and capacities.

From applying any algorithm, entry \((m, \hat{c})\) will contain the maximum attainable profit from using integer quantities of items type \( 1, 2, \ldots, m \) and a knapsack with capacity \( \hat{c} \). For example, assume as input parameters 5 items, a knapsack with maximum capacity 8, and a list of individual profits \((p_i)\) and weights \((w_i)\) for the items. The grid for solving the problem consists of eight columns (total capacity) and five rows (total items). The entries at the 1st row contain the maximum attainable profits if loading only item 1 for different knapsack capacities \( \hat{c} \), the ones in the 2\textsuperscript{nd} row contain the maximum attainable profits after loading items 1 and 2 in the knapsack, and so on. Thus, entry \((4, 7)\) on the grid contains the maximum or optimal profit attainable after loading items 1, 2, 3 and 4 in a knapsack with capacity 7. Now assume that 2 units of item 1, 1 unit of items 2 and 3, and 3 units of item 4 are the optimal quantities to select. The maximum attainable profit for entry \((4, 7)\) is given by:

\[
\max_{\hat{c}} P_{(4,7)} = 2 \times P_{\text{item}1} + 1 \times P_{\text{item}2} + 1 \times P_{\text{item}3} + 3 \times P_{\text{item}4}
\]

However, since the optimal quantities to load of each item are not known \textit{a priori}, the computation above cannot be performed in this straightforward way for all grid entries. The computation is based on an iterative procedure, where an item is loaded at a time and the best loading decisions for the item are recorded and used for taking decisions for the next item to be loaded. The single computations for each algorithm studied are described below.
3.1 Classic Approach

This algorithm is based on the forward knapsack dynamic programming recursion proposed by [9] (equation 1 in Section 2.2.2). The algorithm goes through the grid row-by-row. For every item \( m \), the maximum number of items of type \( m \) that can be loaded given a knapsack with capacity \( \hat{c} \) is computed as \( \lfloor \hat{c}/w_m \rfloor \). A choice for the number of units of product \( m \) to load in the knapsack is notated as \( l \). Given a choice of \( l \) units for product \( m \), a possible profit value for entry \((m, \hat{c})\) is calculated adding the previous maximum attainable profit in row \( m-1 \) and column \( \hat{c}-(w_m*l) \) to the profit from loading \( l \) units of product \( m \), that is, \( l*p_m \). After repeating this computation for all possible values of \( l \) \((l=0,1,\ldots,\lfloor \hat{c}/w_m \rfloor)\), the maximum of these profits denoted as \( f_m(\hat{c}) \) is stored in the grid entry \((m, \hat{c})\).

The following is an example of the computation steps for classic. To compute the maximum attainable profit for row 4 and column 8 in the grid, given item = 4, weight = 3, item 4 value = 30, and knapsack capacity \( \hat{c} = 8 \), the steps taken are as follows:

- Visit row 4 in the grid which represents the profit from using items 1 to 4.
- For capacity 8, compute the total units possible to load for item 4.
  \[total = \lfloor \hat{c}/w_4 \rfloor = 8/3 = 2\]
- Loop through \( l = 0 \) to \( l = 2 \) and compute
  \[result = profit_m*l + profit_{(m-1,\hat{c}-(w_m*l))}\]
  In this example, \( result = 30 * l + profit_{(3,8-(3*1))} \) and therefore,
  \[when \ m = 0, result = profit_{(3,8)}\]
  \[when \ m = 1, result = 30 + profit_{(3,5)}\]
  \[when \ m = 2, result = 60 + profit_{(3,2)}\]
- Compute \( \text{max}() \) of results above and store it in the position (4, 8) in the grid. This maximum represents the maximum attainable profit of using items 1, 2, 3 and 4 in a knapsack with capacity 8.

3.2 Morales Approach

This algorithm is based on the forward knapsack dynamic programming recursion proposed by [11]. A parallel implementation for a distributed framework was later proposed by [16]. Similar to classic, this algorithm also goes through the grid row by row. To compute the maximum attainable profit for entry \((m, \hat{c})\), the algorithm compares the maximum attainable profit of not using the item \( m \) at all, which corresponds to the value stored in row \( m-1 \) at column \( \hat{c} \), and the value from using one more unit of current product \( m \) which is the sum of the maximum attainable result at entry \((m, \hat{c}-w_m)\) and the value of item \( m \), \( p_m \). The maximum of these two quantities is recorded entry \((m, \hat{c})\). Following is an example of the computation steps in Morales. To compute the maximum attainable profit for row 4 and column 8 in the grid given item = 4, weight = 3, item 4 value= 30, and knapsack capacity \( \hat{c} = 8 \):

- Visit row 4 in the grid which represents the profit from using items 1 to 4
- Compute the first result component as, the maximum attainable profit in row 3 and column 8 (result1)
- Compute the second result component, \( result2 = profit_{(4,8-3)} + 30 \).
- Find the maximum of \( result1 \) and \( result2 \), that is,
  \[finalresult = \text{max}(result1, result2)\]
3.3 Use of Dominance in Dynamic Programming

In addition to the two main algorithms described above, this paper also evaluates four algorithmic variants based on the use of dominance rules. These four variants are classic_sdb, classic_mdb, morales_sdb and morales_mdb. All variants are based on the forward knapsack dynamic programming recursion proposed by [9].

classic_sdb and morales_sdb apply single dominance. The values computed for the item that dominates are stored (copied in the rows of the non-dominated items) until there is an item that is non-dominated. For example, if there are three items \(m\), \(m+1\) and \(m+2\) and item \(m\) dominates item \(m+1\) and \(m+2\), the maximum values for item \(m\) are saved in rows \(m+1\) and \(m+2\). Item \(m\) will continue being copied until finding another non-dominated item \(m'\). Alternatively, simple dominance concept can be validated in a pre-processing phase and a recursive algorithm can be run just on the non-dominated items.

classic_mdb and morales_mdb are also based on the forward knapsack dynamic programming recursion proposed by [9] but they employ dominance rules to multiple rows.

4. Parallelization Strategies

In this section, we describe parallel variants of classic and morales. Although parallel implementations for both algorithms exist, we incorporate several enhancements in our implementation, including schemes for reducing synchronization cost and exploiting shared-cache locality on multicore systems.

4.1 Parallelizing classic

We describe the parallel variant of classic with the same notation used in Section 3. We assume data is stored in a two-dimensional grid with \(n\) items and a total capacity of \(C\). An index of an item or row in the array is denoted with \(m\) and capacity or column in the array is denoted with \(\hat{c}\). Threads are again used with a shared memory model. The two dimensional array is shared in memory and accessible by all threads.
4.1.1 Row-by-row decomposition

Dependencies in classic occur between consecutive rows (see Fig. 1(a)). That is, $f[m][\hat{c}]$ depends on values $f[m-1][\hat{c}]$ and $f[m-1][\hat{c}-1], f[m-1][\hat{c}-2] \cdots f[m-1][0]$. There is no dependency within the same row. That is, $f[m][\hat{c}]$ does not depend on $f[m][\hat{c}-1], f[m][\hat{c}-2] \cdots f[m][0]$. This makes parallelization of the algorithm easier to accomplish through a row-by-row decomposition. The row $m$ of the two dimensional grid is divided among the number of threads equally into different sections. At a time each thread works on its own section of the row and fills up the row with computed values of $f[m][\hat{c}]$. When all threads have completed their sections (i.e., row $b$ has been computed completely) then the row is changed to $m + 1$. Thus, synchronization barriers appear at the end of each row. The process is repeated until the two-dimensional grid has been computed completely. This scheme processes the capacities in parallel.

Fig. 2 shows an example execution of classic, parallelized using our strategy. Assume, number of threads = 2, $n=4$ and $C=10$. At index $m = 1$, row 1 is divided into two sections of size five each. Thread 1 gets the first five elements $f[m][1] \cdots f[m][5]$ to compute and thread 2 gets the next five elements $f[m][5] \cdots f[m][10]$. It is possible that thread 1 may complete sooner than thread 2. But in that case, thread 1 still has to wait for thread 2 to finish (i.e., entire row to be completed), before moving on to the next row.

Fig. 3 presents pseudocode for the the section of classic that is parallelized. The inner for loop with capacities is the code that is parallelized. The parallelization can be achieved using a single OpenMP directive.

4.1.2 Incorporating Blocking

The strategy described above, produces threads at a very fine granularity, where each thread works on a single capacity. This type of decomposition can add significant overhead in terms of thread creation and synchronization. To ameliorate synchronization cost and increase parallelism granularity, we introduce blocking to classic. The number of blocks is determined by dividing the total capacity $C$ with the block size. A suitable block size is determined empirically through tuning.

We incorporate blocking in classic by dividing the two-dimensional grid into equal size blocks of capacities. The first block covers $\hat{c} = 1$ to $\hat{c} = block\ size$, the second covers $\hat{c} = block + 1$ to $\hat{c} = 2 \times block$, the third covers $\hat{c} = (2 \times block) + 1$ to $\hat{c} = 3 \times block$ and so on. For every block, all items are traversed from row 1 to row $n$. Within a block the parallel algorithm is the same as explained earlier without blocking. The only difference is that the inner loop starts and ends with a $\hat{c}$ index which corresponds to the block it is in. Thus, this method is akin to splitting the two-dimensional grid into smaller two-dimensional arrays with the same number of items of rows as before with the number of cols or capacities equal to the block
// n = total number of items  
// C = total capacity  
// m = item (item/row index in 2D grid M)  
// \( \hat{c} \) = capacity (capacity/col index in 2D grid M)  
// f.aux[\( \hat{c} \)] \approx f[m][\( \hat{c} - \text{weight}[m] \)] + \text{profit}[m]  

for (m ← 1; m ≤ n; m ++) do  
valueitem ← value[m]  
weightitem ← weight[m]  
vcdw ← vessel \( \text{cap} / \text{weightitem} \)  
for \( \hat{c} ← 0; \hat{c} ≤ C; \hat{c} ++ \) do  
l ← 0  
while l ≤ \( \hat{c} / \text{weightitem} \) do  
if m == 1 then  
result ← valueitem \times l  
else  
result ← (valueitem \times l) + f[m − 1][−(weightitem \times l)]  
end if  
if result > jopt then  
jopt ← result  
end if  
l ++  
end while  
f[m][\( \hat{c} \)] ← jopt  
end for  
end for  

Figure 3. Pseudocode for parallel classic size.  

Fig. 4 illustrates our blocking strategy. We assume a block size = 2, n=4, C=6 and number of threads = 2. The grid is divided into \( 6/2 = 3 \) two-dimensional blocks. The first block computes rows \( m = 1 \) to \( m = 4 \) and \( \hat{c} = 1 \) to \( \hat{c} = 2 \). The second block computes \( m = 1 \) to \( m = 4 \) and \( \hat{c} = 3 \) to \( \hat{c} = 4 \). The first block is computed using two threads in the same manner as explained earlier. The first row is split among the two threads. Thread 1 gets \( f[1][1] \) and Thread 2 gets \( f[1][2] \). After row 1 has been completed, computation on the next row begins. Once all rows are completed the first block is done, at which point the second block commences execution. The process is repeated similarly until all blocks are computed.  

4.2 Parallelizing \textit{morales}  

4.2.1 Naive Parallelization  

Since the dependencies in \textit{morales} emanate from the previous row \textit{and} from previous elements of the current row, we have dependencies carried along both dimensions of the grid. As such, a straightforward row-based or column-based decomposition is not feasible. For this reason, we apply diagonal parallelization strategy. When the loop is skewed to compute elements crosswise, it is possible to exploit parallelism along the diagonals. Our parallelization strategy for \textit{morales} is illustrated in Fig. 5. We apply a standard skewing transformation on the loop that performs the main computation. Each diagonal was split equally between the threads, both working on it at the same time. Once all the threads were completed with the diagonal the loop would move on to the next diagonal and the process would be repeated. Hence the parallelization occurred at the diagonals. The maximum parallelism achieved by the algorithm is the longest center diagonal. This naive parallelization scheme results in slower running times than the
4.2.2 Incorporating Simple Pipelining

The Simple Pipeline Algorithm (SPA) from Morales et al. [16] is used to implement an improved variant of *morales*. The SPA algorithm in [16] is written for a distributed system. We modified it to work on a shared-memory (i.e., multicore) architecture. Our implemented SPA algorithm breaks the problem set of items into stages and within each stage, items are divided and assigned to processing cores as independent threads. The SPA parallelization scheme is outlined in Fig. 6. Assume, number of threads = 2, \( n=4 \), \( C=8 \) and \( m \) is the index for the even sequential version. To address this, we implement several enhancements, as described next.
Items
m=1 to 4

Capacities ĉ =1 to 8

Thread 1
Thread 2

This arrow shows dependency

Thread 1
Thread 2

m=2 from stage 1

m=3
m=4

Stage= 1
Stage= 2

Assuming C=8 and n=4, #threads=2
Then num Stages = 2.

Figure 6. Example execution of morales with SPA

Figure 7. Pseudo code for parallel morales with SPA

// n = total number of items
// C = total capacity
// m = item (item/row index in 2D grid M)
// ĉ = capacity (capacity/col index in 2D grid M)
// f.aux[ĉ] ≈ f[m][ĉ - weight[m]] + profit[m]

for stage ← 1 to (n/numThreads) do
    m ← (stage − l) × numThreads + threadID
    if weight[m] ≤ C then
        f.aux[weight[m]] ← profit[m]
    end if
    for ĉ ← 1 to C do
        // getting input? f[m − l][ĉ]
        if ĉ ≥ weight[m] then
            f[m][ĉ] ← max(f[m − l][Ĉ], f.aux[ĉ])
        else
            f[m][ĉ] ← f[m − l][Ĉ]
        end if
        if ĉ + weight[m] ≤ C then
            f.aux[ĉ + weight[m]] ← f[m][ĉ] + profit[m]
        end if
    end for
    // sending output? f[m][ĉ]
end for
end for

items in the two-dimensional grid, shared among all cores. The formula to calculate m takes the stage, numThreads and thread ID. Number of computation stages = n/numThreads = 2. So in stage 1, thread 1 gets item m = 0 × 2 + 1 = 1 and thread 2 gets item m = 2. In stage 2 thread 1 will get m = 3 and thread 2 gets m = 4. The dependencies in morales imply that to compute f[m][ĉ], values in f[m − 1][ĉ] and f[m][ĉ − weight[m]] must have been already computed. The SPA strategy works based on the observation that if values f[m][ĉ] are computed in increasing order of ĉ, as soon as f[m − 1][ĉ] is available, f[m][ĉ − weight[m]] is also available. In SPA, a processing core covers one item m, which corresponds to a row m in the two-dimensional grid. The thread that computes values of m, stores the values of f[m][ĉ] locally. Therefore, values in f[m][ĉ − weight[m]] + profit[Ĉ] required by the next thread is stored locally by the thread working on item or row m. An auxiliary array is created in each thread to store this values (f.aux in Fig. 7. To get the value of f[m − 1][ĉ] the thread need values from the previous row or item m − 1. It receives this from another thread working on item or row m − 1. Hence, the communication and synchronization that occurs between the threads is
between adjacent rows.

4.2.3 Handling communication between threads

The SPA scheme follows a producer-consumer model where an individual thread serves as both a consumer and a producer. This complicates inter-thread communication. If not orchestrated carefully, communication costs can add significant overhead. Here, we describe the communication optimization implemented in parallel morales with SPA.

The thread that produces values $f[m][\hat{c}]$ needs to consume values $f[m-1][\hat{c}]$. It can only consume the value of $f[m-1][\hat{c}]$ once it has been computed by the thread responsible for it. Until then the thread in charge of row $m$ needs to wait for a signal from the thread of row $m-1$. Once the thread in charge of the row $m-1$ has finished computing the value of $f[m-1][\hat{c}]$ it sends a signal to the waiting thread. In the original SPA algorithm for distributed systems, the data value of $f[m][\hat{c}]$ was itself sent to the respective waiting processor. A ring topology was used where processor $p$ was directly connected to processor $a+1$, with last processor being connected via the root processor to the first processor. The root processor managed the queue of the data messages [16]. This communication model fails through on multicore architectures since the message queue is shared among all cores. Designating a root thread to handle this communication would result in the shared array being perpetually locked. Hence, to adopt SPA for multicore architecture we implement a model of communication that employs non-blocking senders and a receivers. The communication model is implemented using condition variables in Pthreads\footnote{Implementation based on OpenMP-flush and Pthreads-locks caused significant performance loss}. In this model, every element has a condition variable associated with it (instead of a semaphore lock). This condition variable determines access to the shared data structure. The use of condition variables in Pthreads increased the memory requirements for the program and causing some performance loss. This issue is handled next.

4.2.4 Improving Synchronization Cost

The final implementation of morales with SPA is illustrated in Fig. 8. This implementation uses a novel strategy to minimize synchronization and communication costs. In this strategy, threads communicate with each other about the status of the two-dimensional grid by setting a two-dimensional boolean array, consume. The row index of the consume array denotes the threadID and the second column index denotes the capacity $\hat{c}$. A single thread communicates to its adjacent thread. Condition variables are used for signaling and waiting. Locks are used with each condition variable and also at the same time used to secure single access at a time to the consume array. Every thread has one condition variable to use and one lock. Single dimensional arrays consume and cdlock represent this with threadID as their index.

Consider an example with two thread and the algorithm shown in Fig. 8. The consume array consists of size row index of two elements and column index of size of the capacity $\hat{c}$ from 1 to $C$. The consume and cdlock array are of size two each. The consume array is initialized to false. For the first row, thread 1 only produces since the starting values are zero. Hence, $consume[1][\hat{c} : 1 to C]$ is set to true. If thread 1 is working on row $m-1$ then thread 2 is working on row $m$. When thread 1 starts it acquires the lock $cdlock[1]$ and checks to see if $consume[1][\hat{c}]$ is set to true for $f[m-1][\hat{c}]$. This value is true, therefore it is set to false and $cdlock[1]$ is unlocked. Thread 1 then moves on computes $f[m-1][\hat{c}]$ and thread 2 acquires
// \( n \) = total number of items
// \( C \) = total capacity
// \( m \) = item (item/row index in 2D grid \( M \))
// \( \hat{c} \) = capacity (capacity/col index in 2D grid \( M \))
// \( f.aux[\hat{c}] \approx f[m][\hat{c} - weight[m]] + profit[m] \)

for stage ← 1 to \((n/\text{numThreads})\) do
  \( m \leftarrow (\text{stage} - l) \times \text{numThreads} + \text{threadID} \)
  if weight[m] \( \leq C \) then
    \( f.aux[\text{weight}[m]] \leftarrow \text{profit}[m] \)
  end if
  for \( \hat{c} \leftarrow 1 \) to \( C \) do
    // getting input
    if \( \text{consume}[\text{threadID}][\hat{c}] \neq \text{true} \) then
      wait(\( \text{consume}[\text{threadID}], \text{cdlock[threadID]} \))
    end if
    \( \text{consume}[\text{threadID}][\hat{c}] \leftarrow \text{false} \)
    unlock(\( \text{cdlock[threadID]} \))
    if \( \hat{c} \geq \text{weight}[m] \) then
      \( f[m][\hat{c}] \leftarrow \text{MAX}(f[m-l][\hat{c}], f.aux[\hat{c}]) \)
    else
      \( f[m][\hat{c}] \leftarrow f[m-l][\hat{c}] \)
    end if
    if \( m + \text{weight}[m] \leq C \) then
      \( f.aux[\hat{c} + \text{weight}[m]] \leftarrow f[m][\hat{c}] + \text{profit}[m] \)
    end if
  end for
end for

Figure 8. Pseudocode for parallel 	extit{morales} with SPA and synchronization optimization

lock \( \text{cdlock[2]} \) and checks if \( \text{consume[2][\hat{c}]} \) is set to true; it is not and hence thread 2 goes to wait for \( \text{consume[2]} \) and unlocks \( \text{cdlock[2]} \). When thread 1 is done with computing \( f[m-1][\hat{c}] \) it acquires a lock \( \text{cdlock[2]} \) and sets \( \text{consume[2][\hat{c}]} \) to true. It then signals with a condition variable \( \text{consume[2]} \) and unlocks \( \text{cdlock[2]} \). Thread 1 moves on to the next element \( \hat{c} + 1 \) or if it is done with the row \( m - 1 \), then it moves on to row \( m + 1 \). Thread 2 gets the signal \( \text{consume[2]} \), gets out of the wait state and locks \( \text{cdlock[2]} \). It sets \( \text{consume[2][\hat{c}]} \) to false and unlocks \( \text{cdlock[2]} \) and proceeds to compute \( f[m][\hat{c}] \). It then locks \( \text{cdlock[1]} \) and sets \( \text{consume[1][\hat{c}]} \) to true. Finally, thread 2 signals with \( \text{consume[1]} \) and unlocks \( \text{cdlock[1]} \). At any given time two adjacent threads will be working on one row each and the consume array is reused every time and reset by the threads respectively. The order of setting the consume array from true to false with respect to capacity \( \hat{c} \) is important and the fact that it has to be locked whenever changes to it are made.
4.2.5 Incorporating Blocking

The threads in the algorithm depicted in Fig. 8 are communicating and synchronizing one element at a time. The computation done by each thread for $f[m][c]$ is not sufficient to keep processing cores busy for a significant amount of time. As a result, the synchronization overhead dominate the running times. As was done for classic, we address this problem by introducing blocking. Now instead of each thread computing just one element at a time, it is set to compute a block of elements. This is intended to increase the computation workload for each thread and balance out the synchronization overhead. We show an example execution of blocked parallel *morales* in Fig. 9. The modified algorithm with blocking is not shown but is essentially the same except that now a block of elements are computed at a time. If the block size was $K$ then thread 1 computes $K$ elements and then signals thread 2. Thread 2 would now be waiting for $K$ elements of the $m-1$ row to be computed. Once all $K$ elements have been computed, thread 2 is signaled and gets out of wait and computes $K$ elements before it signals the next adjacent thread. The total capacity $C$ is divided by the block size to determine the number of blocks.

5. Experimental Results

5.1 Experimental Framework

5.1.1 Platforms

Each parallel variant is evaluated on three different multicore platforms. Table 1 shows the hardware and software configuration of each experimental platform. The two-core system is based on Intel’s *Conroe* chip, in which the L2 cache is shared between the two cores. The four-core system uses Intel’s *Kentsfield* chip, which has two L2 caches, shared between two cores on each socket. The eight-core platform has a Xeon processor with Hyper threading (HT) enabled, providing 16 logical
Table 2. Procedure to generate $p_i$ and $c_i$ for IKP instances

<table>
<thead>
<tr>
<th>Instance type</th>
<th>$w_i$</th>
<th>$p_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>Random in $[1, R]$</td>
<td>Random in $[1, R]$</td>
</tr>
<tr>
<td>WC</td>
<td>Random in $[1, R]$</td>
<td>Random in $[w_i - R/10, w_i + R/10]$</td>
</tr>
<tr>
<td>SC</td>
<td>Random in $[1, R]$</td>
<td>$p_i = w_i + 10$</td>
</tr>
<tr>
<td>SS</td>
<td>Random in $[1, R]$</td>
<td>$p_i = w_i$</td>
</tr>
</tbody>
</table>

cores. The rest of this paper refers to the two- four- and eight-core platforms as Core2, Quad and 8Core.

5.1.2 Implementation

The parallel variant of classic is implemented with OpenMP, while parallel Morales is implemented with Pthreads. Although, the overhead of an OpenMP implementation may differ from that of a Pthreads implementation, in our study, we did not observe any noticeable differences in overhead between classic and Morales.

Both variants are compiled with GCC version 4.1.2, with default (-O2) optimization settings. For each implementation number of concurrent threads is generally limited to the number of available cores on the target machine. One main thread was used to create and run the worker threads and then wait for them to finish and then end the program. So when 2 threads for Morales are mentioned, these are the two worker threads excluding the main thread. So in total there are three threads running. Similarly, on Quad there are 5 threads in total and 8Core there are 9 threads, including the main thread.

5.1.3 Performance Evaluation

Wall clock time is measured by embedding calls to OpenMP timer routines within the source code. When reporting execution times only the running time for the algorithm itself in the application is reported, and thus excluding any overhead associated with calls to timer routines and file I/O operations. HPCToolkit [1] is used to problem HW performance counters to measure CPU cycles, L1 and L2 cache misses, no of instructions completed and other performance metrics. Use of HW performance counters adds some overhead to the running time. We used HW counter multiplexing to reduce this overhead. The remaining overhead was deemed insignificant for the purposes of this study. To avoid OS jitter, each experimental run is replicated five times and only the consistent lowest values are considered.

5.1.4 Data Set Generation

We use Pisinger’s method [17] for generating IKP instances with pseudo random weights and profits ($w_i$ and $p_i$). This procedure permits generation of uncorrelated (UC), weakly correlated (WC), strongly correlated (SC) or subset-sum knapsack instances (SS) of any size. Table 11 summarizes the methods used for generating $w_i$ and $p_i$ for each instance type. Unless otherwise noted, experimental runs use the default $10000 \times 10000$, UC data set. For this data set random values are selected from $1 : R$, where $R = 100$ for 10000 weights and 10000 profits.

Generated values are unsorted by default. However, for some experiments we vary the sortedness of the data set. In term of sortedness, three different data sets are used:

1. **Unsorted**: no sorting of data values
2. **Sorted by Weight**: The data set was sorted in terms of the items weights in ascending order with the smallest weight on top going down to the largest weight at the bottom.
3. **Sorted by Profit/Weight**: The data set was sorted in terms of the profit
Figure 10. Performance improvement with increasing number of cores

Figure 11. Performance improvement with increasing data sets on 8Core

divided by the weight of each item. The item having the greatest value of profit/weight was on top, then the next value and so on in a descending order.

5.2 Scalability

Fig. 10 shows speedup obtained over the sequential version, for both classic and morales for 2, 4, and 8 cores. This chart reveals that both parallel variants obtain significant speedup over their sequential counterparts. However, classic yields higher speedup than morales on each platform.

Fig. 11 shows performance improvement of classic and morales on 8Core for increasing data set sizes. We observe that both parallel variants achieve significant speedup over the sequential version for all three data set sizes. However, morales yields higher speedup as the size of the data set is increased, obtaining a speedup of a factor six for the 60000 × 60000 data set. Thus, morales appears to be more scalable in terms of the size of the data set but not in terms of the number of processing cores. These experimental results suggests that classic exhibits strong scaling whereas morales exhibits weak scaling.

A somewhat surprising outcome of this set of experiments was that the absolute running time for the sequential algorithm on the 30000 × 30000 data set was less than either of the other two data sets. This anomaly was caused by the choice of the $R$ value in the data set. We investigate and explain this behavior further, later in this section.
5.3 Impact of Blocking Factor and Granularity

The blocking factor controls both the granularity of parallelism and the data locality among concurrent threads. A series of experiments is conducted to evaluate the performance impact of blocking factors for both parallel variants on all three platforms. Results of these experiments are summarized in Figs. 12-14. These figures show performance of \textit{classic} and \textit{morales} on different platforms, as block sizes are varied. Interestingly, we observe a clear performance trend for both \textit{classic} and \textit{morales} on all three platforms. The performance drops significantly for smaller block sizes, picks up as we increase the block size and then drops again when we increase the block size beyond a certain threshold. On \textit{Core2} and \textit{Quad} this threshold is around the $5K$ mark whereas on \textit{8Core} it is around the $2500$ mark. The poor performance for smaller block sizes is speculated to be due to a result of
poor granularity. When block sizes are \( \leq 48 \), concurrent threads are not assigned enough computation to offset the overhead of thread creation and synchronization. On Core2 and Quad, any block size smaller than 48 or larger than \( 5K \) turns out to be a poor choice. It should be noted, that although total cache capacity on Quad is larger than Core2, the available cache per socket is still the same, and thus, the range of good tile sizes appears to be the same for both platforms. On 8Core, for classic even with smaller block sizes there is still speedup. For morales algorithm there is no speed up when the block size is \( < 10 \). Since the speedup on 8Core is greater than Core2 and Quad, we observe a steeper performance curve.

In order to explain this performance pattern due to variations in block sizes, we take a closer look at the memory behavior and CPU utilization of individual threads in each parallel variant. Figs. 15 and 16 show per-thread L1 and L2 cache miss rate for classic on Core2. Figs. 17 and 18 show number of completed instructions
Figure 18. CPU cycle count for parallel classic on Core2

Figure 19. L1 miss rates for parallel morales on Core2

Figure 20. L2 miss rates for parallel morales on Core2

Figure 21. Instructions completed for parallel morales on Core2
and CPU utilization for classic.

Overall we observe that for classic, L1 and L2 miss rates decrease and became fairly constant after block size 1000. The CPU cycles for classic continue to decrease till the block size 1000 is reached and then start to climb back up after block size 2500. Number of instructions completed also increase after block size 1000. The fact that the L1, L2 misses are higher before block size 48 do account for the lower performance. The same goes for CPU cycles and instructions completed when the block size is less than 48. This goes hand in hand with the performance pattern observed in Fig. 12.

Figs. 19 and 20 show per-thread L1 and L2 cache miss rate for morales on Core2. Figs. 21 and 22 show number of completed instructions and CPU utilization for morales. Looking at the L1 cache miss rates for morales, we see that they start high and decrease after block size 5K and become more or less constant. The L2 cache miss rates are fairly uniform across all block sizes. CPU cycles and instructions completed follow a similar trend as L1 miss rates: higher for block sizes is < 48. This matches the same lower bound for the block size observed earlier. However, the upper limit does not follow the same trend exactly. Looking at the L1 and L2 cache miss rates, CPU cycles and instructions completed of the morales one sees that after block size 5000 everything is mostly constant. This does explain why speedup starts to decrease after block size 5K. Unlike classic, there is no noticeable change in CPU cycles, instructions completed or L1 cache misses in thread 1, beyond this point. To explain the performance curve in morales we ran another set of experiments and measured the TLB miss rates for varying block sizes. These experiments revealed, that unlike classic, the performance drop in morales is not caused by excess L1 caches misses but rather misses in the level one data TLB.

5.4 Evaluating Dominance-based Strategies

5.4.1 Dominance and Sortedness

The impact on running times on using the dominance optimization on the algorithms using the default unsorted data set is shown in Table 3. Simple dominance greatly reduced the running time for classic and multiple dominance brought about even more significant improvement. Even with morales both simple dominance and multiple dominance reduced running times. Overall, multiple dominance gave best results for both variants. If the data set is sorted by weight, then execution time for classic with the dominance is greatly reduced. Both simple and multiple dominance produced equivalent running times. morales did not seem to
be affected much with this sorted data set. When the data set is sorted by the profit to weight ratio, \textit{classic} is again affected, while \textit{morales} remains the same. The running time with simple and multiple dominance with \textit{classic} is reduced further. The performance gains obtained through dominance based strategies, is a direct consequence of the algorithms having to process fewer data items. If an item is dominated by another, then instead of computing the entire row, the row would simply be copied from the dominant item row.

5.4.2 Dominance and Data Correlation

Table 4 shows the reduction in the data set search space as a result of using dominance based strategies. The number of rows skipped in computation were identical for \textit{classic} and \textit{morales}. For this reason, we only report numbers for \textit{classic}. We observe that the dominance strategy is most effective when the data is uncorrelated and sorted by $W$. Both single and multiple dominance is able to skip almost all rows in this case. We notice that sorting has a big impact in increasing the number of rows being skipped. For simple dominance, sorting by weight appears much more effective than sorting by profit/weight. It is only in the strongly correlated data set that sorting by profit/weight increased the number of skipped rows. For the SS data set, sorting by profit/weight could not be done because in the data set the weight was equal to the profit of each item. The data correlation (UC, WC, SC, SS) also influences the number of rows being skipped. For multiple dominance, overall the SC data set had the least of the numbers of rows being skipped. Next were WC and then UC. The SS data set type would sometimes be equal to SC or UC. For Simple dominance the SC and SS data set had the least number of rows being skipped, next were WC and then UC. So overall for both dominance types the SC data set type showed the least number of rows being skipped. Also again one can
see that Multiple dominance is a better optimization than simple dominance.

6. Related Work

6.1 Parallelization on multicore

Tan et al. discussed a parallel programming algorithm for a multicore architecture[20]. They presented a scheme to exploit fine grain parallelism and locality of a dynamic programming algorithm with non-uniform dependence on a multicore architecture. The multicore architecture they tested their algorithm on was the IBM Cyclops64 simulator. They proposed that, since this architecture model was an extension conventional out-of-core model, their algorithm solution can be adapted to achieve high performance on a conventional out-of-core model.

Holzmann described a stack slicing algorithm whose application was for multicore model checking. The Stack slicing algorithm tried to achieve an even distribution of work across the available CPUs(load balancing), maximal independence between the work done on different CPUs and minimal communication overhead [13]. The focus was primarily on shared memory systems but could be easily extended to use on cluster computers. The algorithm was a modified and parallelized depth-first search and was compared to the classic depth-first search and proved that an inherently sequential process can be parallelized. Villa et al. discuss the challenges and design choices involved in parallelizing a breadth-first search algorithm on the Cell Broadband Engine multicore processor [22]. The Cell BE is meant for high performance clusters and supercomputers and its memory hierarchy is explicitly managed at software level. They described how they parallelized the Breadth-first search algorithm and by experimentation proved that their method achieved a high level of performance on the Cell BE processor.

Scarpazza et al. look deeper into the parallelization of the Breadth first search algorithm on the Cell BE processor [19]. The paper was a bit similar to the one mentioned before but included more details and experimental results. With the Breadth-first search graph exploration, they proved that it is possible to tame the algorithmic and software development process and achieve, at the same time, an impressive level of performance. They mentioned that explicit management of the memory hierarchy, with emphasis on the local memories of the multiple cores, is a fundamental aspect that needs to be captured by the high level algorithmic design to guarantee portability of performance across existing and future multicore architectures. They also added that the major strength of the Cell BE processor was the possibility of overcoming the memory wall: the user can explicitly orchestrate the memory traffic by pipelining multiple DMA requests to the main memory. This is a unique feature that is not available on other commodity multi-processors, which cannot efficiently handle working sets that overflow the cache memory.

6.2 Parallelization of the IKP

There are fewer papers on parallelizing the IKP than on parallelizing the 0/1 KP. The work mostly closely related to the work presented in this paper is by Morales et al. [16]. These authors present multiple strategies for parallelizing the DP recursion 2 proposed by [11] for the IKP. The results are tested in a distributed framework for transputer and LAN networks using Occam and PVM, respectively. The first algorithm is a simple pipeline algorithm (SPA) that performs a parallelization on the objects. The implementation is on a one-way ring topology with a root processor to facilitate synchronization and administration of the queue of messages.
(computed \( f \) values).

Authors in [16] do not consider blocking as an alternative to improve the performance of SPA but they implemented SPA with single dominance [10] to reduce the high communication cost between processors. The new algorithm is named pipelined algorithm with dominance (PAD). Only non-dominated solutions from a previous row are sent to the next processor to compute 2. A future study may consider to combine dominance and the blocking concept proposed in this research to improve performance for \textit{morales} algorithm.

Paper in [16] also implemented a pipelined algorithm with parallelization on the capacities (PAPC). For a particular knapsack item, each processor computes the \( f \) values for a range of capacities of fixed length \( r \). Dependency may occur among non-adjacent processors and therefore a particular processor may wait for values sent by a processor different to its predecessor. The knapsack items are associated with the iterations or steps of the algorithm. Only after a row of the matrix \( M \) is completely computed, the algorithm proceeds with the computations for the next item (row).

Authors in [3] present a parallel pipelined algorithm for the IKP with time complexity equal to the one in [16], that is, \( \mathcal{O}(nC/q + n) \) where \( q \) is the number of processors. The algorithm was implemented on a ring topology and the speedup resulted asymptotically linear on \( q \). Authors also present a new procedure for the backtracking phase with a time complexity \( \mathcal{O}(n) \), an improvement to \( \mathcal{O}(mc) \), the time complexity of usual strategies.

7. Conclusions and Future Work

This paper explored the issue of algorithmic choice in the context of the integral knapsack problem on multicore architectures. The experimental results reveal that although a row-by-row problem decomposition does not fare well when run sequentially, it exhibits good scalability when run in parallel. The nature of scaling exhibited by \textit{classic} is \textit{strong scaling}, whereas \textit{morales} exhibited \textit{weak scaling}. Thus, when a large number of processing cores are available, \textit{classic} should be the preferred algorithm, whereas when handling large data sets, \textit{morales} is a better choice. The experimental results also showed that blocking factors have a significant impact on performance of each parallel variant. Therefore, to achieve improved performance, it is necessary to select blocking factors through careful analysis and tuning.

The impact of dominance-based strategies on sequential IKP was also explored. Dominance significantly reduced the search space of large data sets and improved running times. This improvement in reducing the search space to smaller data sets would increase the performance of the parallel variants even further. Future research will explore effects on performance of the parallel variants to changes in data set sizes. The data locality aspects of performance will also be examined in more depth.

One point to note is that the \textit{classic} parallel variant was considerably simpler to parallelize and was implemented using OpenMP. Transforming a sequential program to a parallel one using OpenMP usually requires minimal code restructuring compared to other methods like Pthreads. Thus, if programmer productivity is taken into account along with performance gains for a multicore architecture then the \textit{classic} parallel variant is ideal, as this demonstrates a generic method in transforming a sequential algorithm with minimal effort and significant performance gains when run on a multicore systems.
References


