PARALLEL KNAPSACK ALGORITHMS ON MULTICORE ARCHITECTURES

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Hammad A. Rashid, B.S

San Marcos, Texas
May 2010

PARALLEL KNAPSACK ALGORITHMS ON MULTICORE ARCHITECTURES

Committee Members Approved:

Apan Qasem, Chair

Clara Novoa

Mark McKenney

Approved:

J. Michael Willoughby
Dean of the Graduate College

ACKNOWLEDGMENTS

I would firstly like to thank God Almighty without whose help and support nothing would be possible. I would then like to thank Dr. Apan Qasem for all of his help and support. I would also like to thank Dr. Clara Novoa for her help and support as well. Lastly I would like to thank Dr. Mark Mckenney for agreeing to be on my committee and for his insight and help.

This manuscript was submitted on May 12, 2010.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

ABSTRACT

PARALLEL KNAPSACK ALGORITHMS ON MULTICORE ARCHITECTURES

by

Hammad A. Rashid

Texas State University-San Marcos

May 2010

SUPERVISING PROFESSOR: APAN QASEM

Emergence of chip multiprocessor systems has dramatically increased the performance potential of computer systems. Since the amount of exploited parallelism is directly influenced by the selection of the algorithm, algorithmic choice also plays a critical role in achieving high performance on modern architectures. Hence, in the era of multicore computing, it is important to re-evaluate algorithms efficiency 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 algorithms and their parallel implementations, and examines several aspects of performance including speedup and scalability.

CHAPTER 1

INTRODUCTION

It is widely agreed, that the 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 couple of years - perhaps decades. This fundamental shift in processor design technology implies that software plays 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, run-time systems need to schedule concurrent 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 parallelism that can be extracted by the compiler is determined by the initial choice of the algorithm. For example, in the combinatorial optimization field dynamic programming and branch-and-bound algorithms are used to solve optimization problems but these algorithms have different degrees of parallelism and therefore, lead to widely varying performance. Thus, it is important to consider *algorithmic choice* when implementing parallel solutions on current chip multiprocessor (CMP) architectures.

Finding the most suitable algorithm that will deliver high-performance across different architectures and problem sizes has always been a significant challenge,

researched over 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 [7, 26, 5]. However, the emergence of CMP systems adds a new level of complexity. CMP architectures contain one or more levels of cache shared among multiple processing cores. A shared-cache (or memory, in general) poses an inherent trade-off between data locality and parallelism [24]. On one hand, any parallel decomposition will inevitably influence the data access patterns from concurrent threads and consequently affect locality. On the other hand, any transformation for improving locality will impose constraints on parallelism that will affect performance. Thus, when parallelizing an application for CMP architectures, it is imperative to find the right balance between data locality and parallelism. Since algorithmic choice dictates the amount of exploited parallelism and data locality, it plays a key role in obtaining high-performance on CMP systems.

This research studies the impact of algorithmic choice on the performance of parallel algorithms for solving the integral knapsack problem (IKP) under a dynamic programming (DP) approach. Two different DP algorithms are studied. IKP is very relevant in combinatorial optimization because it directly models practical situations such as capital budgeting [22], cutting stock [9, 10] cargo loading problems [4, 14] and scheduling of batch processors [18]. Furthermore, IKP's appear as sub-problems in set-partitioning formulations for multi-dimensional cutting stock [11], crew scheduling and generalized assignment problems [3]. IKP's have also contributed to the generation of minimal cover induced constraints and in

the development of coefficient reduction procedures for strengthening bounds for general integer programming (IP) problems [16]. This relationship between IKP's and other IP's has motivated great interest for developing efficient IKP algorithms.

Several factors make IKP a suitable target for evaluating the impact of algorithm choice. IKP algorithms under a DP approach are amenable to different types of parallelism. For example, some IKP algorithms can be parallelized in a pipelined fashion [2] and in other algorithms the central loop lends itself to a data parallel decomposition. Also most implementations of IKP exhibit data locality that can be exploited through compiler transformations.

The primary goal of this study is to understand the performance trade-offs from choosing a particular type of DP algorithm for solving an IKP instance. Specifically this research makes the following contributions:

- a quantitative analysis of performance for two DP algorithms is presented. To the best of our knowledge, no previous work has looked at performance issues for IKP algorithms on multicore architectures.

- a key *tunable* parameter is identified which can be used to significantly enhance performance of both parallel algorithms.

The rest of this document is organized as follows: chapter 2 provides background on the IKP; chapter 3 describes the sequential algorithms used and also discusses optimizing them with dominance; chapter 4 disscusses the parallel implementations of the sequential algorithms; chapter 5 presents experimental results and analysis;

chapter 6 discusses related work on parallelization of IKP; and finally, chapter 7 provides conclusions.

CHAPTER 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 $x_1, ..., x_n$, where x_j represents the number of jth 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)* [16]. 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, $x_j = 1, ..., b_j$. Following is the IP formulation for the UKP: Maximize

$$\sum_{j=1}^{n} p_j x_j \qquad \text{(UKP)}$$

subject to

$$\begin{aligned} \sum_{j=1}^{n} w_j x_j &\leq C \qquad\qquad j = 1, ..., n \\ x_j &\geq 0 \text{ and integer}, \quad j = 1, ..., n \end{aligned}$$

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 [16]. 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 exactly KP's are branch and bound (B&B) [16] and dynamic programming (DP) [15, 1, 2]. 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 [16] presented a DP forward recursion (see Eq. 2.1) 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.

$$f_m(\hat{c}) = max\{f_{m-1}(\hat{c} - lw_m) + lp_m\}$$
$$l \ integer, \ \ 0 \leq l \leq \lfloor \hat{c}/w_m \rfloor \tag{2.1}$$

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 [17]. Authors in [11] mention that this recursion (see Eq. 2.2) is more efficient than the one in Eq. 2.1. The recursion is given by:

$$f_m(\hat{c}) = max\{p_m + f_m(\hat{c} - w_m), f_{m-1}(\hat{c})\} \tag{2.2}$$

$m = 1, 2, ..., n$ and $\hat{c} = 0, ...C$. Equation 2.2 selects between loading or not a unit of product m. This recursion involves less operations than the one in equation 2.1 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 [2]. 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 2.1 and 2.2) 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 [1].

2.2.3 Hybrid approaches

The work in [20] is the first hybrid approach for solving IKP's. The algorithm significantly outperforms all existing algorithms for solving the problem. The algorithm success comes from embedding B&B into a DP framework, applying multiple bounds including a new stronger one, and exploiting many IKP properties such as dominance relations and periodicity.

2.3 Schemes for Reducing the Search Space

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 $w_i >= w_j$ and $p_i <= p_j$.

In [16] *multiple dominance*(md) is introduced. It states that object i is multiple dominated by object j if and only if $\lfloor w_i/w_j \rfloor \geq p_i/p_j$. This dominance relation means that j dominates i when the profit from all the objects 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 [1].

CHAPTER 3

EVALUATING SEQUENTIAL KNAPSACK ALGORITHMS

3.1 Dynamic Programming Algorithms

This research implements the DP forward recursions 1 [9] and 2 [11] provided in Section 2.2. In the rest of the paper, these two algorithms are referred as `classic` and `morales`. Both algorithms use a two dimensional matrix M of n rows that represent the items and C columns that represent the knapsack capacities. The indexes $(m, \hat{c})$ will be used to represent any (row, column) pair in the grid. The knapsack capacity is in terms of weight but it could be any other relevant problem dimension that needs to be used efficiently such as volume, length, etc. The goal of the algorithms is to compute the maximum attainable profit (value) from selecting any item and capacities combination.

From applying any algorithm, entry $(m, \hat{c})$ will contain the maximum attainable profit(value) from using integer quantities of items type $1, 2, m$ and a knapsack with capacity $\hat{c}$. For example, assume as input parameters 5 items, a knapsack with maximum capacity 8 kg, and a list of individual profits (p_i) and weights (w_i) for the items. The grid for solving the problem consists of 8 columns (total capacity) and 5 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 2nd 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 kg. 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 :

Maximum profit for entry (4,7) = 2 * (profit of item 1) + 1 * (profit of item 2) + 1*(profit of item 3) + 3 * (profit of item 4).

However, since the optimal quantities to load of each item are not known "a priori", the computation above cannot be performed in this straightforward way for all grid entries. The computation will be based on an iterative procedure where on 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 load. The single computations for each algorithm studied are described below.

3.1.1 Classic Algorithm

***rrks2* algorithm** This algorithm is based on the forward knapsack dynamic programming recursion proposed by [9] (equation 2.1 in section 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, $(l = 0, 1, , \lfloor \hat{c}/w_m \rfloor)$, 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 *rrks2* algorithm. Compute the maximum attainable profit for row 4 and column 8 of the grid given item 4 weight= 3 kg, item 4 value= 30, and knapsack capacity $\hat{c} = 8$ kg.

- Go to 4th row 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 = $\lfloor \hat{c}/w4 \rfloor = 8/3 = 2$
- Perform the following loop from $l = 0$ to $l = 2$

  for l=0 to 2 do: result = $(p_m * l)$ + profit value row $[m - 1]$ column $[\hat{c} - (w_m * l)]$

  In this example, result = (30*l + profit value row [3] column[8 - (3 * l)]) and therefore:

  - when m=0, result = profit row [3] column [8];
  - when m=1, result= 30 + profit row[3] column [8-3]
  - when m=2, result= 60 + profit row[3] column [8 - 6]
- Compute the maximum of the results above and store it in the 4th row, 8th column of the grid. This maximum represents the maximum attainable profit of using items 1, 2, 3 and 4 in a knapsack with capacity 8 kg.

Fig. 3.1 shows the pseudo code for the algorithm.

```
// 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)
// l =  (letter L ) , number of items m possible with ĉ

for m: =1 to n do
    for ĉ := 1 to C do

            l:=0
          while (l <= (ĉ/weight[m]))

                  if (l equal to 1)
                      result = profit[m] * l;
                  else
                      result = (profit[m] * l) + f[m-1][ ĉ - (weight[m] * l)];

                   if (result > maxvalue)
                         maxvalue := result;

               l++;

           endwhile

           f[m][ĉ]= maxvalue;

   endfor
endfor
```

Figure 3.1: Pseudo code of the Classic algorithm(rrks2)

Fig. 3.2 illustrates the dependence between entries of the grid for the computation of the maximum attainable profits.

3.1.2 Morales Algorithm

***rrks3* algorithm**. This algorithm is based on the forward knapsack dynamic programming recursion proposed by [11] and parallelized by [17] in a distributed framework. The algorithm also goes through the grid row by row. To compute a maximum attainable result for entry $(m, \hat{c})$, the algorithm compares the maximum attainable result(profit) of not using the item m at all, which corresponds to the

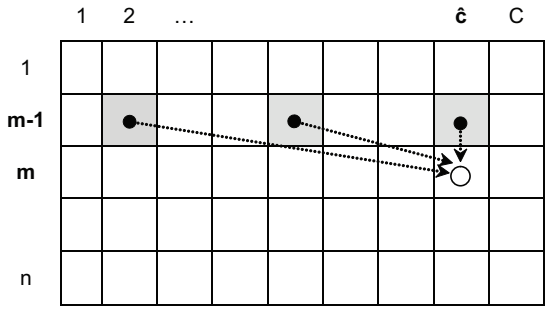

Figure 3.2: Dependence between entries for Classic Algorithm (rrks2)

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 $(m, \hat{c})$.

The following is an example of the computation steps for *rrks3* algorithm.Compute the maximum attainable profit for row 4 and column 8 of the grid given item 4 weight= 3 kg, item 4 value= 30, and knapsack capacity $\hat{c} = 8$ kg.

- Go to the 4th row in the grid that represents the value from using items 1 to 4.
- Get $result1$, the maximum attainable profit in row 3 and column 8.
- Compute $result2$, by getting the profit in row 4 and column $(8 - w_m)$ and adding it to the profit of item m, p_m. In the example, result2 = profit row [4] [8-3] + 30
- Find the maximum of $result1$ and $result2$, that is,

  $finalresult = maxresult1, result2$

- Store $finalresult$ in row 4 at column 8. This maximum represents the maximum attainable profit of using items 1, 2, 3 and 4 in a knapsack with capacity 8 kg.

Fig. 3.3 shows the pseudo code for the algorithm.

```
// 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)


for ĉ:= 1 to C do
      f[0][ĉ]:= 0;

for m: =1 to n do
   for ĉ := 1 to C do
        if  (ĉ < weight[m])
            f[m][c] := f[m-1][ ĉ]
        else
            f[m][ĉ] := max {(f[m][ĉ -weight[m]] + profit[m]) , f[m-1][ĉ]};

   endfor
endfor
```

Figure 3.3: Pseudo code of the Morales Algorithm (rrks3)

Fig. 3.4 illustrates the dependence between entries of the grid for the computation of the maximum attainable profits.

3.1.3 The use of Dominance

3.1.3.1 Classic Algorithm

***rrks2sdb* algorithm.** This is the algorithm based on the forward knapsack dynamic programming recursion proposed by [9] and enhanced with the simple dominance concept explained in section 2.3 Schemes for reducing the search space.

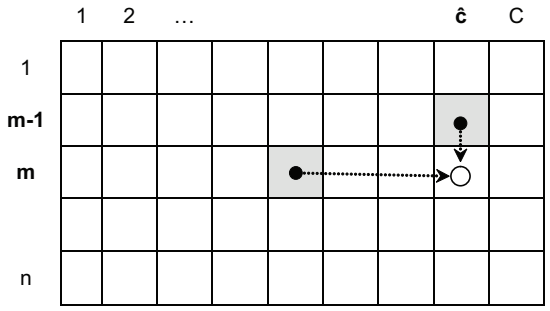

Figure 3.4: Dependence between entries for Morales (rrks3)

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 results 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, the simple dominance concept can be validated in a pre-processing phase and the recursive algorithm *rrks2* can be just run on the non-dominated items.

***rrks2mdb* algorithm.** This is the algorithm based on the forward knapsack dynamic programming recursion proposed by [9] enhanced with the multiple dominance concept explained in the section 2.3 Schemes for reducing the search space. The steps and comments in *rrks2sdb* apply to *rrk2mdb* just updating the simple dominance criterion to multiple

3.1.3.2 Morales Algorithm

***rrks3sdb* algorithm.** This is the algorithm based on the forward knapsack dynamic programming recursion proposed by [11] and parallelized by [17] enhanced with the simple dominance concept explained in the section 2.3 Schemes for reducing the search space. 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 results non-dominated. Alternatively, the simple dominance concept can be validated in a pre-processing phase and the recursive algorithm *rrks3* can be run just on the non-dominated items.

***rrks3mdb* algorithm.** This is the algorithm based on the forward knapsack dynamic programming recursion proposed by [11] and parallelized by [17] enhanced with the multiple dominance concept explained in Section 2.3 Schemes for reducing the search space. 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 results non-dominated. Alternatively, the multiple dominance concept can be validated in a pre-processing phase and the recursive algorithm *rrks3* can be run just on the non-dominated items.

3.2 Data set generation

This research uses the procedure in [19] for generating instances with pseudo random profits and weights. This procedure permits to generate uncorrelated (UC), weakly correlated (WC), strongly correlated (SC) or subset-sum knapsack instances

(SS) of any size. Next table summarizes the way w_j and p_j are generated for each type of instance.

Table 3.1: Procedure to generate p_j and c_j for IKP instances

Instance type	w_j	p_j
UC	Random in $[1, R]$	Random in $[1, R]$
WC	Random in $[1, R]$	Random in $[w_j - R/10, w_j + R/10]$
SC	Random in $[1, R]$	$p_j = w_j + 10$
SS	Random in $[1, R]$	$p_j = w_j$

The main default dataset used was the UC type with random values selected from 1:**R** where **R**=100 for 10000 weights and 10000 profits. The file representing this dataset was called **data.dat**. The data in this file is **unsorted** by default.

3.3 Results of running the dominance optimization on the classic and morales sequential algorithms

The tests were run on a Core 2 Duo Machine which is described in chapter 5. There were two compilers that were used to compile separate executions of the algorithms, the Intel Compiler (ver 10.1) and the G++ compiler (ver 4.1.2).

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. The running time is given in seconds and minutes.

The main default dataset used was the UC type with random values selected from 1:**R** where **R**=100 for 10000 weights and 10000 profits. The file representing this dataset was called **data.dat**. The data in this file is unsorted by default.

For many of the tests the dimensions of 10000 items x 10000 capacity was used.

Besides the UC random dataset type, other datasets of WC, SC and SS were also used. These are described in section 3.2.

For every data set three forms of it were used.

1. **Unsorted**

2. **Sorted by Weight:** The dataset 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 dataset was sorted in terms of the profit 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.

3.3.1 Impact of Sorting

The impact of the running times on using the dominance optimization on the algorithms using the default unsorted dataset is shown in table 3.2.

The Simple dominance greatly reduced the `classic` (rrks2)algorithm's running time and the multiple dominance brought about even more significant improvement. Even with the `Morales` (rrks3) algorithm both simple dominance and multiple dominance reduced running times. Overall the multiple dominance optimizations gave the best results with both algorithms.

If the same dataset was sorted by weight from before, then the running times of the `classic` (rrks2) algorithm with the dominance was greatly reduced. Both the

Table 3.2: Run times using Algorithms with dominance

Data set : data.dat : 10000 items x 10000 capacity **Unsorted** Core2 Machine		
Sequential Algorithms	Runtime in secs (G++ Compiler)	Runtime in secs (Intel compiler)
Classic		
rrks2	204.167	91.3446
rrks2sdb	25	11.8045
rrks2mdb	6.25	2.85197
Morales		
rrks3	1.488	0.453699
rrks3sdb	0.877	0.369372
rrks3mdb	0.81	0.35578

simple and multiple dominance produced equivalent running times. This is shown in table 3.3. However, without dominance the performance of the `classic` algorithm (rrks2) was reduced when using a dataset sorted in this manner. The running time of algorithm when compiled using g++ increased from 204.167 sec to 250.907 sec. The `Morales` algorithm did not seem to be affected much with this sorted dataset.

When the dataset was sorted by the profit/weight the`classic` (rrks2) algorithm was again affected. This is seen in table 3.4. The `morales` algorithm remained the same for the most part as before. The running times with simple and multiple

Table 3.3: Run times using Algorithms with dominance

Data set : sdata.dat : 10000 items x 10000 capacity , Core2 Machine

Sorted Data Set by Weight

Algorithms	Runtime in Secs (G++ Compiler)	Runtime in secs (Intel compiler)
Classic		
rrks2	250.907	112.229
rrks2sdb	3.02108	1.34349
rrks2mdb	3.02	1.34825
Morales		
rrks3	1.48572	0.454638
rrks3sdb	0.808467	0.3556
rrks3mdb	0.808747	0.355684

dominance with the `classic`(rrsk2) algorithm were reduced further from before in table 3.3. Both had similar running times. Without dominance the result with the two compilers was somewhat different. With the g++ compiled version of rrks2 the running time was reduced to 184 sec from 204 sec (unsorted dataset) while the Intel compiler version had an increased running time of 108.887 sec from 91.34 sec (unsorted dataset).

The running times of the algorithms due to dominance were reduced because there were lesser data items to deal with in the n (items) x C (total capacity) matrix. If an item was dominated by another, then instead of computing the entire row, the row would simply be copied from the dominant item row. This was measured by counting the number of rows skipped (from full computation) or copied from the dominant item. Another way for the rows to be simply copied over and skipped from computation is when the item weight is greater than the capacity and

Table 3.4: Run times using Algorithms with dominance

Data set : sdatavdw.dat : 10000 items x 10000 capacity , Core2 Machine

Sorted Data Set by Profit/Weight

Algorithms	Runtime in Secs (G++ Compiler)	Runtime in secs (Intel compiler)
Classic		
rrks2	184.913	108.887
rrks2sdb	1.8611	0.908475
rrks2mdb	1.86115	0.910027
Morales		
rrks3	1.49148	0.454513
rrks3sdb	0.807288	0.35678
rrks3mdb	0.806284	0.355291

this is already part of the original algorithm without dominance. This form of row skipping was not measured. Only the rows skipped due to dominance were recorded.

Table 3.5 shows the reduction in the data set search space by using the dominance optimization by showing the number of rows skipped in computation. The **data.dat** dataset was used.

From the results in table 3.5 it can be seen that the number of rows skipped in computation are the same for `classic` and `morales`. This is because the same optimization was applied identically to both algorithms. The reason for the running times being the same for simple dominance and multiple dominance, when the dataset is sorted (by weight, by profit/weight) can be explained by table 3.5. The running times of both rrks2sdb (simple dominance) and rrks2mdb (multiple dominance) were the same because the number of rows skipped by the optimizations

Table 3.5: Number of rows skipped in computation using Algorithms with dominance

Number of Rows Skipped in Computation			
Data set : data.dat : 10000 items x 10000 capacity Unsorted , UC (Uncorrelated)			
G++ Compiled code			
Sequential Algorithm	**Sorted by weight**	**Sorted by profit/weight**	**Unsorted**
Classic			
rrks2sdb	9994	9999	9035
rrks2mdb	9994	9999	9964
Morales			
rrks3sdb	9994	9999	9035
rrks3mdb	9994	9999	9964

was the same. This can now clearly be seen from table 3.5. The same conclusion is for the `morales` algorithm.

Theoretically however multiple dominance should skip more rows than simple dominance, as it does a more thorough comparison than simple dominance. This is proven from the results in the next of figures starting from Fig. 3.6 .

3.3.2 Using Different types of datasets and Sizes with sorting

The figures starting from table 3.6 show additional results of using dominance with datasets of the UC, WC, SC and SS type. Different datasets of 10000 items with 10000 capacity and 5000 items with 5000 capacity were used. The value of **R** for these datasets was changed to 30000. So for example, in the UC type dataset the items and profits would be randomly selected from a range 1:30000. The number of rows skipped in computation in the dataset for each algorithm were recorded.

Table 3.6: Number of rows skipped in computation using Algorithms with dominance

Data set: 5000 items x 5000 capacity			
R: 30000			
G++ Compiled code			
UC: Uncorrelated			
Sequential Algorithm	**Sorted by weight**	**Sorted by profit/weight**	**Unsorted**
Classic			
rrks2sdb	4992	18	2459
rrks2mdb	4998	51	4977
Morales			
rrks3sdb	4992	18	2459
rrks3mdb	4998	51	4977

The figures starting from Fig. 3.13 show the results of 4 different randomly generated datasets in one table. Each dataset had 10000 items with 10000 capacity and **R**=30000.

3.3.3 Summary

One can see that sorting had a big impact in increasing the number of rows being skipped. For Simple dominance Sorting by weight seemed a lot more effective than Sorting by Profit/Weight. Except for table 3.5, in most of all the other figures it decreased the number of rows being skipped with the UC and WC dataset type. It was only in mostly the SC dataset type that it increased the number of skipped rows. For the SS dataset type Sorting by Profit/Weight could not be done because in the dataset the weight was equal to the profit of each item and so it was not

shown in the results in all the figures.

For Multiple dominance Sorting by weight overall seemed a lot more effective for the different datasets. With the UC dataset type in some cases the Sorting by Profit/Weight gave better optimization than Sorting by Weight and in some cases it gave a lower reduction of the rows being skipped.

The dataset type (UC, WC,SC,SS) gave different results of the rows being skipped. For Multiple dominance overall the SC dataset type had the least of the numbers of rows being skipped. Next were WC and then UC. The SS dataset type would sometimes be equal to SC or UC. For Simple dominance the SC and SS dataset had the least number of rows being skipped, next were WC and then UC. So overall for both dominances the SC dataset type showed the least number of rows being skipped.

Also again one can see that Multiple dominance is a better optimization than simple dominance.

Table 3.7: Number of rows skipped in computation using Algorithms with dominance

Data set: 5000 items x 5000 capacity			
R: 30000			
G++ Compiled code			
WC: Weakly correlated			
Sequential Algorithm	**Sorted by weight**	**Sorted by profit/weight**	**Unsorted**
Classic			
rrks2sdb	4822	137	175
rrks2mdb	4997	4999	4992
Morales			
rrks3sdb	4822	137	175
rrks3mdb	4997	4999	4992
Data set: 5000 items x 5000 capacity			
R: 30000			
G++ Compiled code			
SC: Strongly correlated			
Sequential Algorithm	**Sorted by weight**	**Sorted by profit/weight**	**Unsorted**
Classic			
rrks2sdb	587	587	1
rrks2mdb	587	587	1
Morales			
rrks3sdb	587	587	1
rrks3mdb	587	587	1

Table 3.8: Number of rows skipped in computation using Algorithms with dominance

Data set: 5000 items x 5000 capacity		
R: 30000		
G++ Compiled code		
SS: P=W (profit=weight)		
Sequential Algorithm	**Sorted by weight**	**Unsorted**
Classic		
rrks2sdb	587	1
rrks2mdb	4999	3196
Morales		
rrks3sdb	587	1
rrks3mdb	4999	3196

Table 3.9: Number of rows skipped in computation using Algorithms with dominance

Data set: 10000 items x 10000 capacity			
R: 30000			
G++ Compiled code			
UC: Uncorrelated			
Sequential Algorithm	**Sorted by weight**	**Sorted by profit/weight**	**Unsorted**
Classic			
rrks2sdb	9989	29	4961
rrks2mdb	9999	9999	9975
Morales			
rrks3sdb	9989	29	4961
rrks3mdb	9999	9999	9975

Table 3.10: Number of rows skipped in computation using Algorithms with dominance

Data set: 10000 items x 10000 capacity			
R: 30000			
G++ Compiled code			
WC: Weakly Uncorrelated			
Sequential Algorithm	**Sorted by weight**	**Sorted by profit/weight**	**Unsorted**
Classic			
rrks2sdb	9729	291	346
rrks2mdb	9993	9999	9986
Morales			
rrks3sdb	9729	291	346
rrks3mdb	9993	9999	9986

Table 3.11: Number of rows skipped in computation using Algorithms with dominance

Data set: 10000 items x 10000 capacity			
R: 30000			
G++ Compiled code			
SC: Strongly Uncorrelated			
Sequential Algorithm	**Sorted by weight**	**Sorted by profit/weight**	**Unsorted**
Classic			
rrks2sdb	1445	1445	0
rrks2mdb	1445	1445	0
Morales			
rrks3sdb	1445	1445	0
rrks3mdb	1445	1445	0

Table 3.12: Number of rows skipped in computation using Algorithms with dominance

Data set: 10000 items x 10000 capacity		
R: 30000		
G++ Compiled code		
SS: P=W (profit=weight)		
Sequential Algorithm	**Sorted by weight**	**Unsorted**
Classic		
rrks2sdb	1445	0
rrks2mdb	1445	1
Morales		
rrks3sdb	1445	0
rrks3mdb	1445	1

Table 3.13: Number of rows skipped in computation using Algorithms with dominance

There are 4 different randomly generated datasets each having: 10000 items x 10000 capacity													
R: 30000													
G++ Compiled code													
UC: Uncorrelated													
Sequential Algorithm		**Sorted by weight**				**Sorted by p/w**				**Unsorted**			
		Dataset 1	**Dataset 2**	**Dataset 3**	**Dateset 4**	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dateset 4**	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dateset 4**
Classic													
rrks2sdb		9987	9991	9998	9995	24	18	367	51	4820	4732	4977	4863
rrks2mdb		9998	9998	9999	9999	9999	9998	9999	9999	9981	9983	9982	9981
Morales													
rrks3sdb		9987	9991	9998	9995	24	18	367	51	4820	4732	4977	4863
rrks3mdb		9998	9998	9999	9999	9999	9998	9999	9999	9981	9983	9982	9981

Table 3.14: Number of rows skipped in computation using Algorithms with dominance

There are 4 different randomly generated datasets each having:													
10000 items x 10000 capacity													
R: 30000													
G++ Compiled code													
WC: Weakly Uncorrelated													
Sequential Algorithm		**Sorted by weight**				**Sorted by p/w**				**Unsorted**			
		Dataset 1	**Dataset 2**	**Dataset 3**	**Dateset 4**	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dateset 4**	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dateset 4**
Classic													
rrks2sdb		9998	9992	9995	9994	9999	3192	4261	2492	7849	440	1810	1242
rrks2mdb		9998	9994	9997	9997	9999	9999	9999	9999	9973	9992	9971	9995
Morales													
rrks3sdb		9998	9992	9995	9994	9999	3192	4261	2492	7849	440	1810	1242
rrks3mdb		9998	9994	9997	9997	9999	9999	9999	9999	9973	9992	9971	9995

Table 3.15: Number of rows skipped in computation using Algorithms with dominance

There are 4 different randomly generated datasets each having: 10000 items x 10000 capacity													
R: 30000													
G++ Compiled code													
SC: Strongly Uncorrelated													
Sequential Algorithm		**Sorted by weight**				**Sorted by p/w**				**Unsorted**			
		Dataset 1	**Dataset 2**	**Dataset 3**	**Dateset 4**	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dateset 4**	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dateset 4**
Classic													
rrks2sdb		1476	1468	1494	1515	1476	1468	1494	1515	0	1	0	0
rrks2mdb		1476	1468	1494	1515	1476	1468	1494	1515	0	1	0	0
Morales													
rrks3sdb		1476	1468	1494	1515	1476	1468	1494	1515	0	1	0	0
rrks3mdb		1476	1468	1494	1515	1476	1468	1494	1515	0	1	0	0

Table 3.16: Number of rows skipped in computation using Algorithms with dominance

10000 items x 10000 capacity									
R: 30000									
G++ Compiled code									
SS: P=W (profit=weight)									
Sequential Algorithm		**Sorted by weight**				**Unsorted**			
		Dataset 1	**Dataset 2**	**Dataset 3**	**Dateset 4**	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dateset 4**
Classic									
rrks2sdb		1476	1468	1494	1515	0	1	0	0
rrks2mdb		1478	1468	1497	9999	3	3	2	6874
Morales									
rrks3sdb		1476	1468	1494	1515	0	1	0	0
rrks3mdb		1478	1468	1497	9999	3	3	2	6874

CHAPTER 4

PARALLELIZATION OF KNAPSACK ALGORITHMS

4.1 Morales Parallel Algorithm

All algorithms were written in C++ and compiled using gcc 4.1.2.

4.1.1 The Sequential algorithms used for Parallelization

The sequential algorithms with dominance described in chapter 3 were not used for parallelization. Only the original algorithms `classic` **(rrks2)** and `morales` **(rrks3)** were used. The reason being is that the dominance optimization reduces the Search space or dataset size. So essentially a smaller dataset is worked on by the algorithm. Larger datasets are needed to better assess the performance of the parallel algorithms.If the parallel algorithms give better performance with larger datasets compared to their sequential equivalents, then they would perform well with smaller datasets as well. When the dominance optimization is applied to both sequential and parallel algorithms both their running times would be reduced , since they are working on a smaller dataset. So if the parallel algorithms out perform their sequential equivalents with larger datasets then they should out perform them when dominance is applied as well.

4.1.2 Naive Parallelization

Since the dependency of the sequential `morales` algorithm was from the previous row and from the previous elements of the current row a diagonal loop approach was considered. Any element of the 2 dim array at a certain time depended on a single element right above it from the previous row and one to the left of it from the current row. So if the loop was run diagonally instead of row by row then these previous elements would always be available(computed and ready from before) whenever the current element is reached.

A for loop was written can goes row by row and translates the row indexes and col indexes to the respective diagonal as it runs through the loop. So a loop was created in this way to traverse the diagonals. 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 parellization occurred at the diagonals. This is illustrated in figure 4.1.

The maximum parallelism achieved by the algorithm would be the longest center diagonal. The algorithm was called *rrks3-p1* and it's running times were slower than the sequential `morales` *(rrks3)*. Table 4.1 shows the comparison. These times are from running the algorithm on the Core2 machine which is described in the experimental results chapter 5. The dataset used was the default *data.dat*. It was a UC type dataset with 10000 items and 10000 total capacity. This could have been because of the overhead of computing the diagonal indexes and that maximum

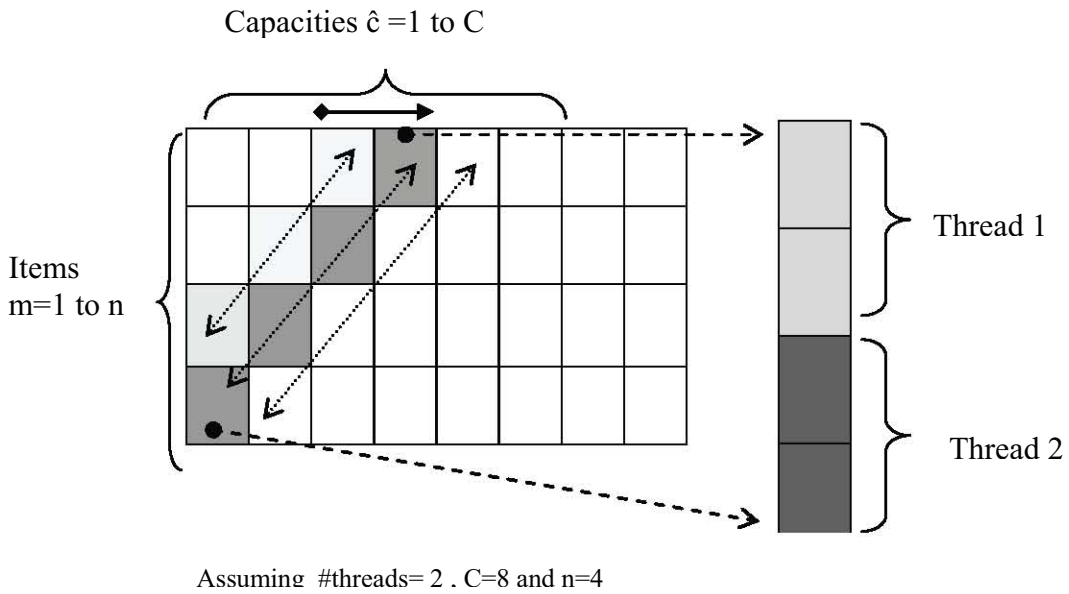

Figure 4.1: Parallelization of the Morales Sequential Algorithm using a Diagonal Approach

parallelism was achieved only at the center diagonal. Because this algorithm ran slower ,attempts were then made to some how implement and adapt the simple pipeline algorithm (SPA) presented in [17] which had proven to be faster than the sequential `morales` for multicore architectures. Later on in the research a parallel implementation of the `morales` algorithm, with the help of this SPA algorithm proved to be faster than the sequential `morales` algorithm and was made the default algorithm for experimentation.

4.1.3 Adapting the SPA algorithm

The Simple Pipeline Algorithm (SPA) from the paper [17] was used to come up with a parallel implementation of the sequential morales algorithm. The SPA algorithm was written for a distributed system. It was modified to work on a shared

Table 4.1: Running times of the Parallel Morales Diagonal Algorithm

Core2 Machine		
data.dat : 10000 items x 10000 capacity		
No of Threads	**rrks3**	**Rrks3-p1**
1	**1.644s**	3.83 s
2		2.165 s
4		2.620 s
8		3.061s
10		3.109s

memory architecture.

The SPA algorithm broke the problem set of the items into stages and within each stage it divided the items to calculate among each of the processors (of the distributed system). In our case threads were used to do the same job as the processors. The outline of the algorithm can be seen in Figure 4.3. The problem consists of a total of n items and a total vessel capacity of C.

In the algorithm m is the index for the items in the two dimensional array of n items and C capacities. The two dimensional array is f[n][C]. This array is shared and accessible by all threads.

The formula to calculate m takes the stage, numthreads and threaded. The threadID is the id of the thread used by the system and it starts from 1 onwards, not zero. The arrays weight and profit contain the weight and profit of the

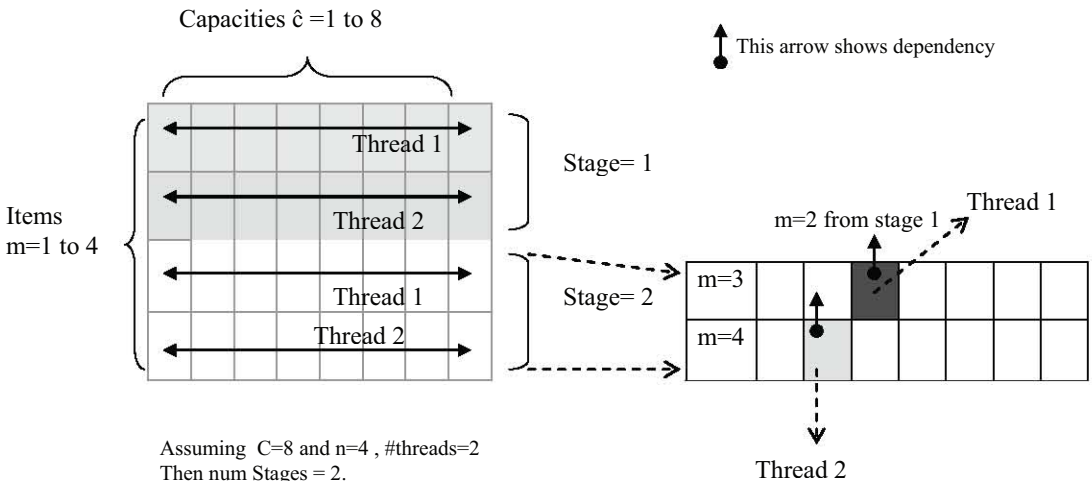

Figure 4.2: The way the SPA Algorithm runs

respective items.

E.g if n=4 , C=8 , numthreads=2. Then the number of stages will be n/numthreads = 4/2 = 2. The threadIDs will be 1 and 2. So in stage 1 , thread 1 will get the item m= 0 * 2 + 1= 1 and thread 2 will get item m=2. In stage 2 thread 1 will get b=3 and thread 2 will get m=4.

As seen earlier there is a dependency in the `morales` sequential algorithm. In order to calculate a value of f[m][$\hat{c}$] of the two dimensional grid, the values of f[m-1][$\hat{c}$] and f[m][$\hat{c}$-weight[m]] must be present. So there is dependency from the previous row and with in the current row itself.

The SPA algorithm states that if the values f[m][$\hat{c}$] are computed in increasing order of $\hat{c}$, as soon as f[m-1][$\hat{c}$] is available, f[m][$\hat{c}$ - weight[m]] is available too [17].

In the SPA algorithm a processor (or thread in our case) covers one item m. This item would be a row m in the two dimensional grid. Since the thread computes the values of the m item row it stores the values computed of f[m][$\hat{c}$] locally. So the

```
// 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/NUM_THREADS) do

        begin

        m:=(stage-l) * NUM_THREADS + threadID

        if (weight[m] <= C)
                f.aux[weight[m]] := profit[m];

        for ĉ:=1 to C do

                begin

                // getting input ? f[m-l][ĉ];


                ------------------------------------

                 if ĉ >= weight[m] then

                        f[m][ĉ] := max { (f[m-l][x], f.aux[ĉ] };
                else
                        f[m][ĉ] := f[m-l][x];

                if(ĉ + weight[m]]<= C) then
                        f.aux[ĉ +weight[m]] := f[m][ĉ] + profit[m];

        ----------------------------------------

        // sending  output? f[m][ĉ];


    end inner for;

end outer for;
```

Figure 4.3: The Pseudo code of the SPA Algorithm

part f[m][$\hat{c}$ - weight[m]] + profit[b] required by the sequential algorithm is stored locally by the thread working on item or row m. This is stored in the f.aux array. An element of the f.aux array represents the value of the f[m][$\hat{c}$ - weight[m]] + profit[m] equation.

e.g weight[2]=3 , profit[2]=10 , m=2, $\hat{c}$=6

- for f[2][6]= max (f[1][6], f.aux[6])

- f.aux[6]= f.aux[3+weight[2]]=f[2][3]+profit[2];

- so f[2][6]= max (f[1][6], f[2][3] + profit[2]) ,when replaced by symbols

- f[m][$\hat{c}$]= max (f[m-1][$\hat{c}$], f[m][$\hat{c}$-weight[m]]+ profit[m])

To get the value of f[m-1][$\hat{c}$] the thread would need values from the previous row or item $m-1$. It receives this from another thread working on item or row $m-1$. So the communication and synchronization that occurs between the threads is between adjacent rows.

4.1.4 Handling communication between threads

In the algorithm a thread is both a consumer and producer. The thread to compute or produce the value f[m][$\hat{c}$] needs to consume the value 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 would have 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 the data value of f[m][$\hat{c}$] was itself sent to the respective waiting processor. A ring topology was used where processor a was connected to processor a+1 , processor a + 1 was connected to processor a+ 2 and so on. The last processor would be connected via the root processor to the first processor. The root processor would manage the queue of the data messages [17].

In this case it is controlling access to the shared array by the threads. The goal is to simulate a non-blocking sender and a receiver that blocks.

Initial attempts to simulate the SPA algorithm's non-blocking sender and blocking receiver were quite unsuccessful. All initial algorithms gave the same correct optimal results as the sequential morales version but their running times were too slow.

The first attempt to simulate this was done using OpenMP flush mechanism. What would happen would be that one thread that was waiting for a result would be constantly polling the shared array using flush and refreshing it's view of the main memory. The algorithm ran slowly for 2 threads but when the number of threads increased the performance dropped significantly and became a lot slower. This could be because they were now more threads polling the shared array, increasing the CPU cycles. This algorithm was called **rrkspmor2**. The second attempt was made with Pthreads and locks. All of the shared array was locked. So basically they were n items x C total capacity locks. Every [row,col] element of the array had a separate associated lock with it. The initial row would be unlocked to start the algorithm. There would then be a sequence of one thread unlocking the element of the next row when it was done computing it while the other thread

would be waiting on a lock for this element. The only way the other thread could get a lock on the element was if the previous associated thread (that computed the element) unlocked the element for it. This algorithm was called **rrkspmor-pth4**. An openmp version of this algorithm was made and it was called **rrkspmor4**. Both of these algorithms were extremely slow mainly most of the overuse of an equal number of locks to the elements of the shared array. Another attempt was made using condition variables in pthreads. Instead of every element of the shared array having a lock associated with it , it has a condition variable instead. In pthreads every condition variable needs a lock associated with it so this algorithm took a lot more memory than the previous ones. This algorithm was called **rrkspmor-pth3**. Apart from memory issues it was very slow as well. It took 2m27.452s with 2 threads to solve a dataset of size 2500 items with 10000 capacity. The sequential `morales` **(rrks3)** solved it in 0.437s.

Table 4.2 shows the running times of these initial algorithms. The dataset used was called *d10.dat*. It was a UC type dataset with 10000 items and 10000 total capacity. The sequential `morales` **(rrks3)** is also included as a comparision.

4.1.5 Improving Synchronization Cost

The final implementation of the SPA algorithm that gave the best running times is shown in Fig. 4.4. This algorithm ran faster and was called **testpth7** and became the default `morales` parallel algorithm for experimentation. For the rest of this thesis this algorithm will be known as the `morales` parallel algorithm. Learning from the previous attempts a new strategy was devised for the threads to

Table 4.2: The running times of the initial algorithms to implement the SPA Algorithm

Core2 Machine **d10.dat : 10000 items x 10000 capacity**						
No of Threads	**rrkspmor2**	**rrkspmor3**	**rrkspmor4**	**rrkspmor-pth3**	**rrkspmor-pth4**	**rrks3**
1	21.934s	7m39.750s	11m18.413s	run out of memory	7m18.427s	**1.644s**
2	18.510s	9m8.444s	13m8.747s	same as above	9m44.002s	
4	1m47.588s	13m4.767s	18m51.604s	same as above	12m28.804s	
8	2m57.521s	14m40.935s	22m13.394s	same as above	14m5.755s	
10	2m57.390s	14m45.516s	22m57.585s	same as above	14m19.031s	

communicate with each other with the least overhead of synchronization and communication. This is how it was implemented.

The threads communicate to each other about the status of the two dimensional array by setting a two dimensional boolean array called consume. The row index of the consume array denotes the threadID and the second column index denotes the capacity $\hat{c}$. A single thread will communicate to it's adjacent thread. Pthreads is used to implement the threads. 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 consumec and cdlock represent this with threadID as their index.

e.g if you have threads 1, 2 and 3 then $1 \rhd 2 \rhd 3$ and $3 \rhd 1$.

So the same ring style topology is used. In the code in figure 4.4 this is shown as the statements to compute nextthread.

Lets give an example between two threads. The consume array will be of size row index of 2 elements (2 threads) and col index of the size of the capacity $\hat{c}$:1 to C. The consumec and cdlock array will be of size 2 each.

The two dimensional consume array is set to false. For the first row or item thread 1 only produces since the starting values are zero. So the consume[1][$\hat{c}$:1 to C] array is set to true. If thread 1 is working on row $m-1$ then thread 2 would be 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 the f[m-1][$\hat{c}$] element. It is true so it sets it to false and unlocks cdlock[1]. It moves on and computes f[m-1][$\hat{c}$] and Thread 2 acquires the lock cdlock[2] and checks if consume[2][$\hat{c}$] is set to true. It is not and It goes to wait for consumec[2] and unlocks cdlock[2].

Whenever thread 1 is done with computing the element f[m-1][$\hat{c}$] it acquires a lock cdlock[2] and writes the consume[2][$\hat{c}$] element to true. It then signals with a condition variable consumec[2] and unlocks cdlock[2]. Thread 1 moves on in the for loop 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 consumec[2] , gets out of the wait state and locks cdlock[2]. It sets consume[2][$\hat{c}$] to false and unlocks cdlock[2] and proceeds to compute the value of f[m][$\hat{c}$]. It then sets to lock cdlock[1] and sets consume[1][$\hat{c}$] to true (remember it was set to false by thread 1). It then signals with consumec[1]

and unlocks cdlock[1].

At a 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 respective to the capacity $\hat{c}$ is important and the fact that it has to be locked whenever changes to it are made.

4.1.6 Blocking

The threads in the algorithm in figure 4.4 are communication and synchronizing one element at a time. The computation done by each thread for the element f[m][$\hat{c}$] is not much however. What ends up happening is that the algorithm runs slower than the sequential counterpart. This is because the synchronization costs are more than the computation. In order to remove this problem blocking was used. Now instead of the thread computing just one element at a time , it is set to compute a block or set of elements. This is intended to increase the computation work load for each thread and balance out the synchronization over head. 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 10 then thread 1 would compute 10 elements and after that signal thread 2. Thread 2 would now be waiting for 10 elements of the $m-1$ row to be computed. Once all the 10 elements have been computed then Thread 2 is signaled and gets out of wait and computes 10 elements before it signals the next adjacent thread. The total capacity C is divided by the block size to determine the number of blocks. This is illustrated in fig. 4.5. As chapter 5 shows, block size has a significant impact on the performance of morales.

```
// 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/NUM_THREADS) do
   m = (stage-l) * NUM_THREADS + threadID
   if (weight[m] ≤ C) then
      f.aux[weight[m]] := profit[m]
   end if
   for ĉ = 1 to C do do
      // getting input? f[m-l][ĉ]
      lock(cdlock[threadID])
      if (consume[threadID][ĉ] ! = true) then
         wait(consume[threadID], cdlock[threadID])
      end if
      consume[threadID][ĉ]=false
      unlock(cdlock[threadID])
      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 (m + weight[m]] ≤ C) then
         f.aux[ĉ +weight[m]] := f[m][ĉ] + profit[m]
      end if
      // sending output? f[m][ĉ]
      int nextthread=0
      if (threadID == (NUM_THREADS-1)) then
         nextthread=0
      else
         nextthread=threadID+1
      end if
      lock(cdlock[nextthread])
      consume[nextthread][ĉ]=true
      signal(consumec[nextthread])
      unlock(cdlock[nextthread])
   end for
end for
```

Figure 4.4: Final Parallel implementation of `morales` using SPA

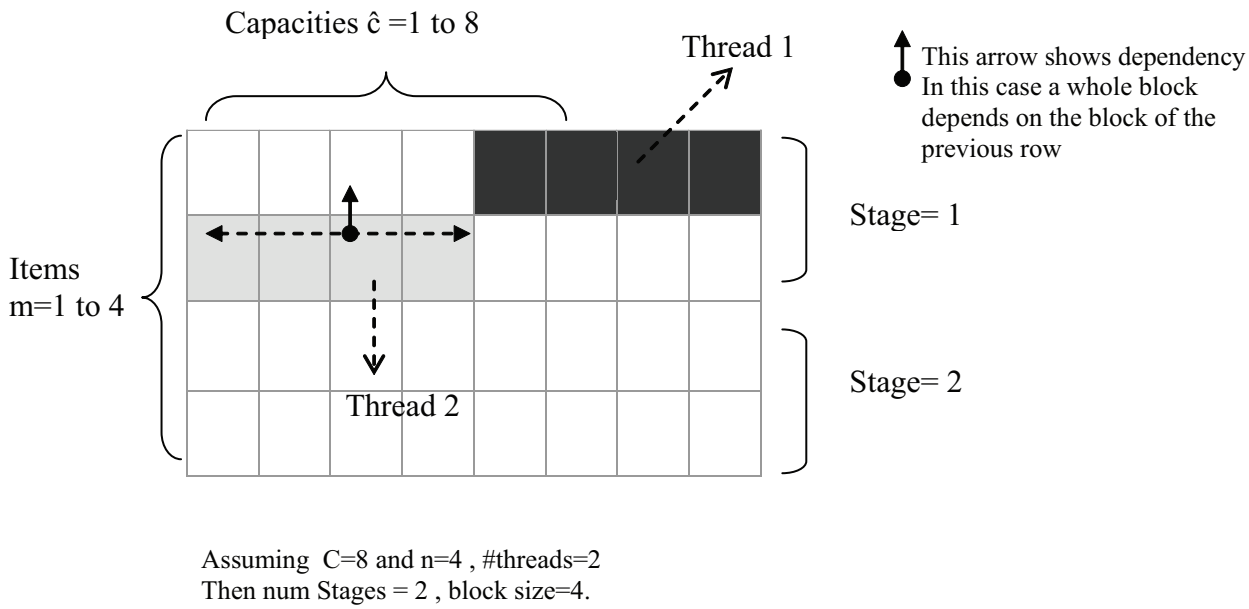

Figure 4.5: The implementation of Blocking in the Morales Parallel Algorithm

4.2 Classic Parallel Algorithm

The same two dimensional grid is used with n items and a total capacity of C. And index of an item or row in the array is denoted with m and a capacity or col 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. Openmp is used to parallelize the algorithm instead of pthreads.

The dependency in the `classic` sequential code is only between the different rows. 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]....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]....f[m][0]. This makes the parallelization of the algorithm easier to accomplish on a row by row basis.

The row m of the two dimensional grid is divided among the number of threads

equally into different sections. At a time each thread is working on it's own section of the row and filling up the row with computed values of f[m][$\hat{c}$]. When all the threads have completed their sections (that is when the row b has been computed completely) then the row is changed to m+1. So the synchronization barrier is at the end of the row.

The process is repeated till the two dimensional grid has been computed completely. This would be a parallelization on the capacities or columns.

e.g They are 2 threads, n=4 and C=10.

At index m=1 , the row 1 is divded into 2 sections of size 5 each. Thread 1 will get the first five elements f[m][1]...f[m][5] to compute and Thread 2 will get the next five elements f[m][5]...f[m][10]. It is possible that Thread 1 might complete sooner than Thread 2. But in this case Thread 1 still has to wait for Thread 2 to finish, that is for the entire row to be completed before moving on to the next row. Fig. 4.6 illustrates this.

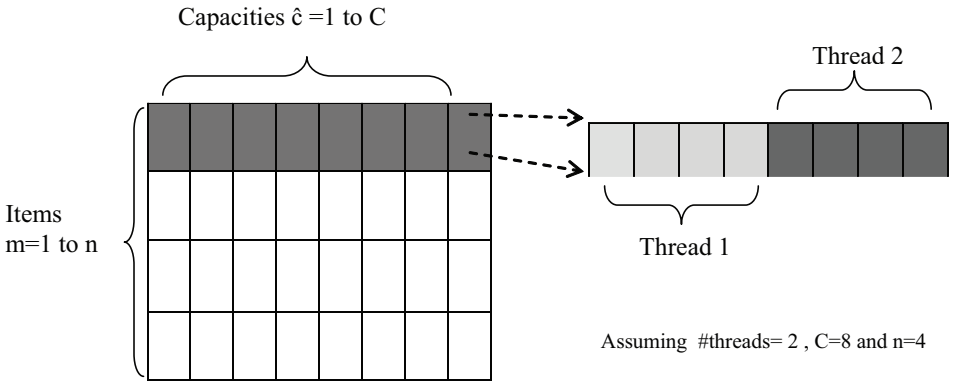

Figure 4.6: The way the Classic Parallel Algorithm works

The c++ code in Figure 4.7 shows the section of the `classic` sequential

algorithm that is parallelized. The inner for loop with the capacities is the section that is parallelized. A single OpenMP statement $pragmaompparallelfor$ is used right before the for loop to parallelize the algorithm.

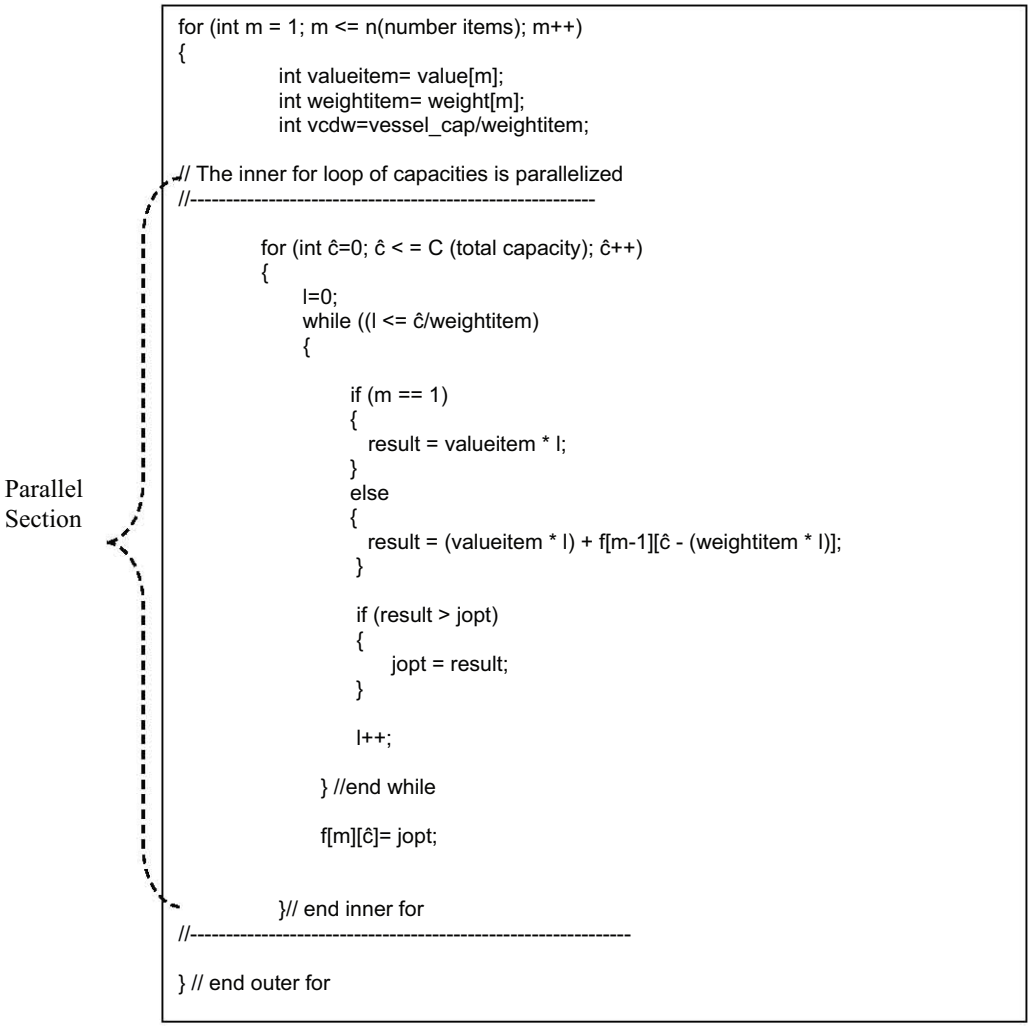

Figure 4.7: The Code of the Classic Parallel Algorithm

4.2.1 Blocking

Blocking has been added to the `classic` parallel algorithm. The number of blocks are determined by dividing the total capacity C with the block size. The two

dimensional grid is divided into equal size blocks of capacities or columns. The first block would be $\hat{c}$=1 to $\hat{c}$=block, the second would $\hat{c}$=block+1 to $\hat{c}$=2*block , the third would be $\hat{c}$=(2*block) +1 to $\hat{c}$=3*block and so on. For every block all the items or rows are traversed from row 1 to row n(number of items). Within a block the parallel algorithm is the same as explained earlier without blocking. The only difference is that the inner for loop of capacities starts from and ends with a $\hat{c}$ index that is respective to the block it is in. So it is more like splitting the two dimensional grid into different smaller two dimensional arrays with the same number of items of rows as before with the number of the cols or capacities equal to the block size. This is demonstrated in Fig. 4.8.

e.g blocksize=2 , n=4, C=6, num of threads=2

So number of blocks = 6/2 = 3. The two dimensional array of 4 rows and 6 cols is thus divided into 3 two dimensional separate arrays or blocks.

- The first block has the same 4 rows (m=1 to 4) and cols $\hat{c}$=1 to $\hat{c}$=2

- The second block has the same 4 rows (m=1 to 4) and cols $\hat{c}$=3 to $\hat{c}$=4

- The third block has the same 4 rows (m=1 to 4) and cols $\hat{c}$=5 to $\hat{c}$=6

The first block is completed computed using the two threads the same way as explained earlier with the classic parallel algorithm without blocking. The row m=1 will split among the two threads. Thread 1 will get f[1][1] and Thread 2 will get f[1][2]. Then after the row m=1 has been completed the next row m=2 is started. Thread 1 will get f[2][1] and thread 2 will get f[2][2] and process repeats till all the rows are completed. Once all the rows are completed the first block is done.

After the first block is calculated the second block is started. As before the block will start with row m=1 to m=4. This time when row m=1 , thread 1 will get f[1][3] and thread 2 will get f[1][4]. The process is repeated similarly till all the rows are computed.

Then finally the last or third block is computed in the same manner as the previous blocks. At row m=1 , thread 1 will get f[1][5] and thread 2 will get f[1][6].

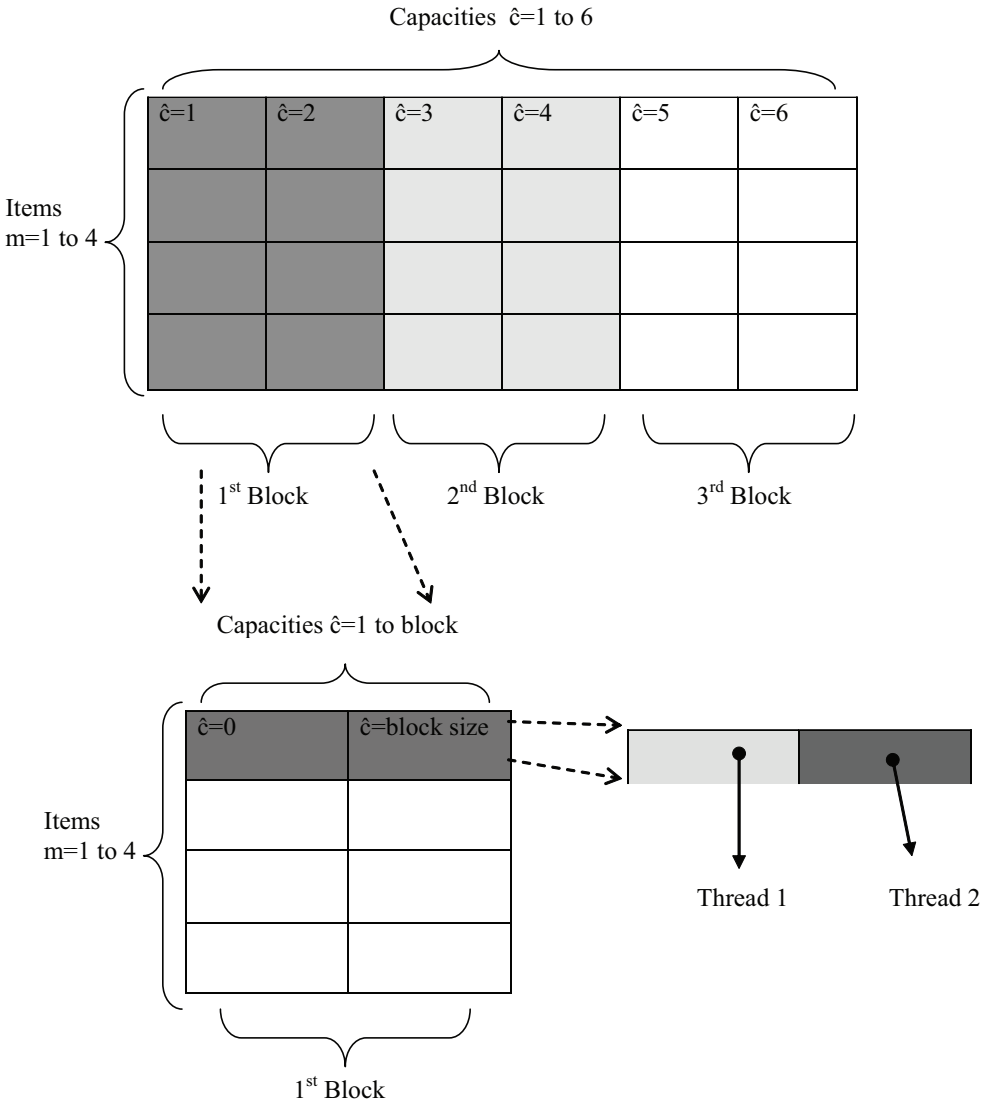

Figure 4.8: Blocking in the Classic Parallel Algorithm

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 Experimental Framework

Each parallel variant is evaluated on three platforms:

1. 2.33 GHz Intel Core 2 Duo E6550 **(Core2, dual core)**. *Core2* contains 2 physical cores. It has a 4 MB L2 Cache shared by the two cores. The system has 2 GB of RAM.

2. 2.40 GHz Intel Core 2 Quad Q6600 **(Quad, 4-core)** system. The *Quad* has 4 physical cores. It has two 4MB L2 caches. Each of the L2 cache is shared between two cores in each socket. The system has 4 GB of RAM.

3. For our scalability study, we also run experiments on an 8 logical core system **(8-core)**. The system has a Xeon 2.53 Ghz E5540 processor with Hyper threading (HT) enabled. It consists of 4 physical cores with HT enabled giving it a total of 8 logical cores. It has a smart cache of 8 MB. The system has 24 GB of RAM.

The classic parallel algorithm is implemented using OpenMP, while the morales version is implemented with pthreads as stated earlier.

Both variants are compiled with GCC version 4.1.2, with the default (-O2) optimization settings. The 2.5 version of Opemmp is used which is already present

with gcc 4.1.2 just like pthreads.

To avoid system jitter, each experimental run is replicated five times and only the consistent lowest values are considered. In some cases, we exclude outlier values, when it is obvious that the extra long running time is due to operating system interference. Wall clock time (in secs, mins) 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.

For each implementation number of concurrent threads was limited to the number of available cores on the target machine.

The HPCToolkit profiler with Hardware PAPI counters was used to measure the CPU cycles, L1, L2 cache misses and instructions completed when running the algorithms. Measuring these parameters increases the wall clock time of running the algorithm by a small amount. Also all of the four parameters cannot be measured at once but only two at a time. This means that one can measure L1, L2 misses in one run and CPU cycles and instructions completed in another run.

Each parallel variant is run with two data sets :

1. The default **data.dat** dataset. It is the UC type with random values selected from 1:R where R=100 for 10000 item weights and 10000 item profits. The data in this file is unsorted. A dimension of 10000 items x 10000 capacity is used.

2. The **ds60-30k.dat** dataset. It is the UC type with random values selected from 1:R where R=30000 for 60000 item weights and 60000 item profits. The data is unsorted. A dimension of 30000 items x 30000 capacity is used. Since they are 60000 items in the set only the first 30000 items are used.

5.2 Speedup and Scalability

Table 5.1: Performance improvement with increasing number of cores

Data set : data.dat : 10000 items x 10000 capacity	Parallel Algorithms Speed up over Sequential Counterparts	
#Cores Machines with same #Threads	Classic	Morales
2	1.803597056	1.608285012
4	3.05874578	2.397716988
8	6.923402524	4.921166928

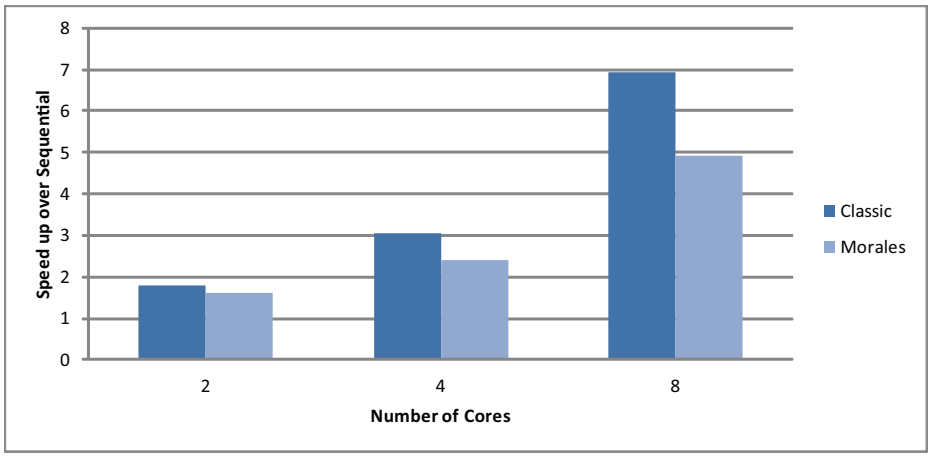

Figure 5.1: Performance improvement with increasing number of cores, Graph from Table 5.1

Table 5.1 and Fig. 5.1 show speedup obtained over the sequential version, for both `classic` and `morales` for 2, 4 and 8 cores.. The data set used was data.dat with 10000 item x 10000 capacity. This chart reveals that both parallel variants obtain significant speedup over their sequential counterparts. However,`classic`

yields higher speedup than `morales`.

5.3 Impact of Blocking Factor and Granularity

The following figures show the running times of both parallel `classic` and `morales` with the variation in block sizes for each of the three machines.

5.3.1 Using the Data set : data.dat : 10000 items x 10000 capacity.

5.3.1.1 Core2 with 2 threads:

Running times and Speed up with varying block sizes

Figures. 5.2 to 5.3 show the impact of the running times and speed up when varying the block sizes, using the data.dat dataset.

Table 5.2: Impact of block size on performance of parallel `classic` on *Core2* using data.dat : 10000 items x 10000 capacity.

Core2 : 2 threads **data.dat : 10000 items x 10000 capacity**			
Classic			
Block size	**Parallel Running time in Seconds**	**Sequential running time in seconds**	**Speed up**
		178.963	
2	463.221		0.386345
10	194.208		0.921502
50	118.467		1.510657
100	107.636		1.662669
200	102.153		1.751911
250	100.672		1.777684
500	99.2256		1.803597
1000	100.228		1.785559
2000	104.258		1.71654
2500	106.429		1.681525
5000	117.904		1.51787
10000	140.847		1.27062

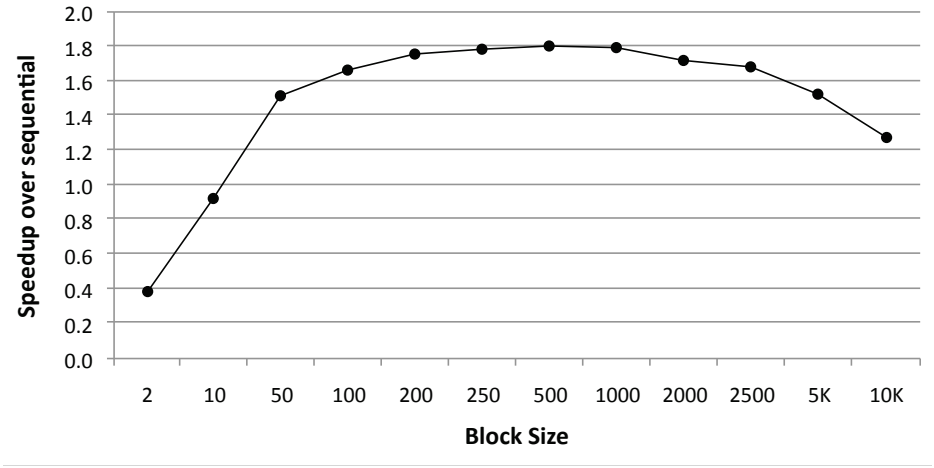

Figure 5.2: Impact of block size on performance of parallel `classic` on *Core2* using data.dat : 10000 items x 10000 capacity, Graph from Table 5.2

Table 5.3: Impact of block size on performance of parallel `morales` on *Core2* using data.dat : 10000 items x 10000 capacity

Core2 : 2 threads **data.dat : 10000 items x 10000 capacity**			
Morales			
Block size	**Parallel Running time in Seconds**	**Sequential running time in seconds**	**Speed up**
		1.475	
2	16.3163		0.0904
10	4.34781		0.339251
50	1.318		1.11912
100	1.08978		1.353484
200	1.00122		1.473203
250	0.982552		1.501193
500	0.949565		1.553343
1000	0.928422		1.588717
2000	0.918092		1.606593
2500	0.917126		1.608285
5000	0.946239		1.558803
10000	1.83768		0.802642

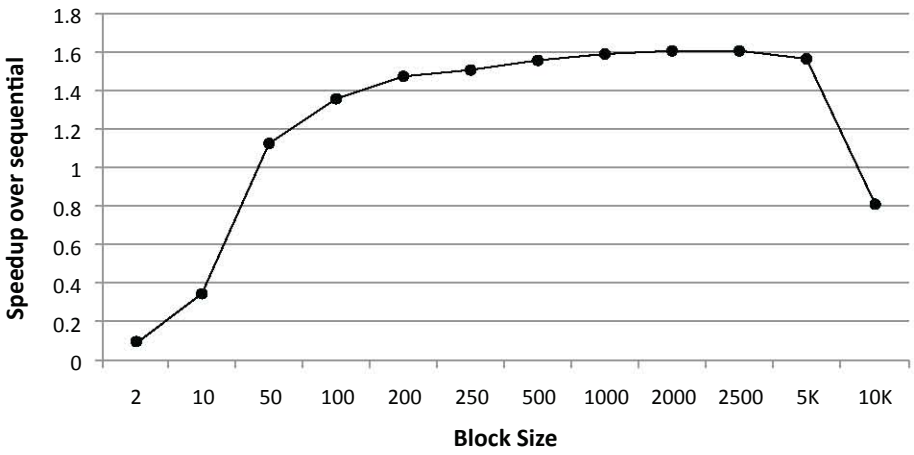

Figure 5.3: Impact of block size on performance of parallel `morales` on *Core2* using data.dat : 10000 items x 10000 capacity, Graph from Table 5.3

Classic parallel algorithm analysis for Core2 with the data.dat (10000 x10000) dataset using 2 threads:

Figures. 5.4 to 5.11 show the L1,L2 cache misses, CPU cycles and Instructions completed for the `classic` parallel algorithm. A breakdown of the results for each individual thread are shown as well.

Table 5.4: Thread 1 L1, L2 Misses on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity

Block size	L1 Misses	L2 Misses	Running time of classic parallel algorithm
10	2920000000	578000000	208.303
50	1470000000	188000000	121.808
100	1280000000	123000000	109.507
200	1130000000	54185418	103.371
250	1060000000	47794779	101.887
500	975000000	27652765	100.16
1000	854000000	15851585	101.038
2000	614000000	9250925	105.048
2500	420000000	8030803	107.449
5000	154000000	4920492	118.939
10000	31793179	3080308	141.828

Table 5.5: Thread 2 L1, L2 Misses on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity

Block size	L1 Misses	L2 Misses
10	2580000000	1710000000
50	1420000000	400000000
100	1250000000	185000000
200	1160000000	105000000
250	1130000000	80958095
500	1130000000	39853985
1000	1180000000	19691969
2000	1370000000	10121012
2500	1550000000	8070807
5000	1780000000	4630463
10000	1950000000	3210321

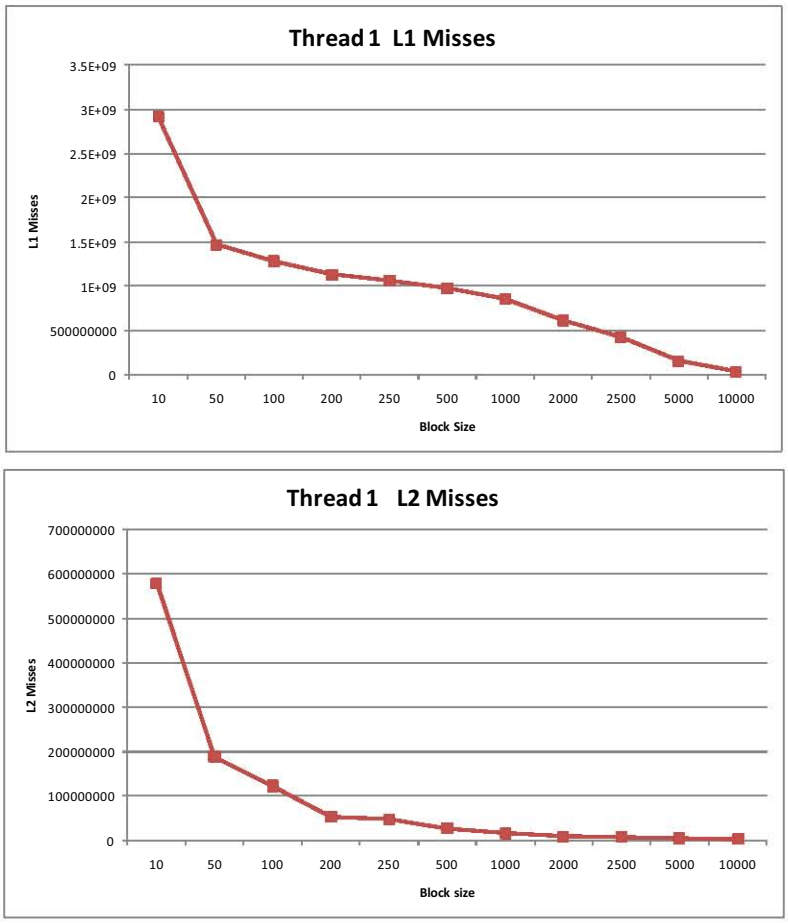

Figure 5.4: Thread 1 L1, L2 Misses on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.4

Table 5.6: Thread 1, Thread 2 Instructions completed on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity

Block size	Thread 1 INS	Thread 2 INS	Total Instructions Completed
10	5.19E+10	43600000000	9.55E+10
50	4.06E+10	42900000000	8.35E+10
100	3.78E+10	30200000000	6.80E+10
200	3.50E+10	31700000000	6.67E+10
250	3.39E+10	28200000000	6.21E+10
500	3.32E+10	28400000000	6.16E+10
1000	3.67E+10	33200000000	6.99E+10
2000	3.97E+10	38700000000	7.84E+10
2500	3.91E+10	43600000000	8.27E+10
5000	3.61E+10	50800000000	8.69E+10
10000	3.43E+10	39700000000	7.40E+10

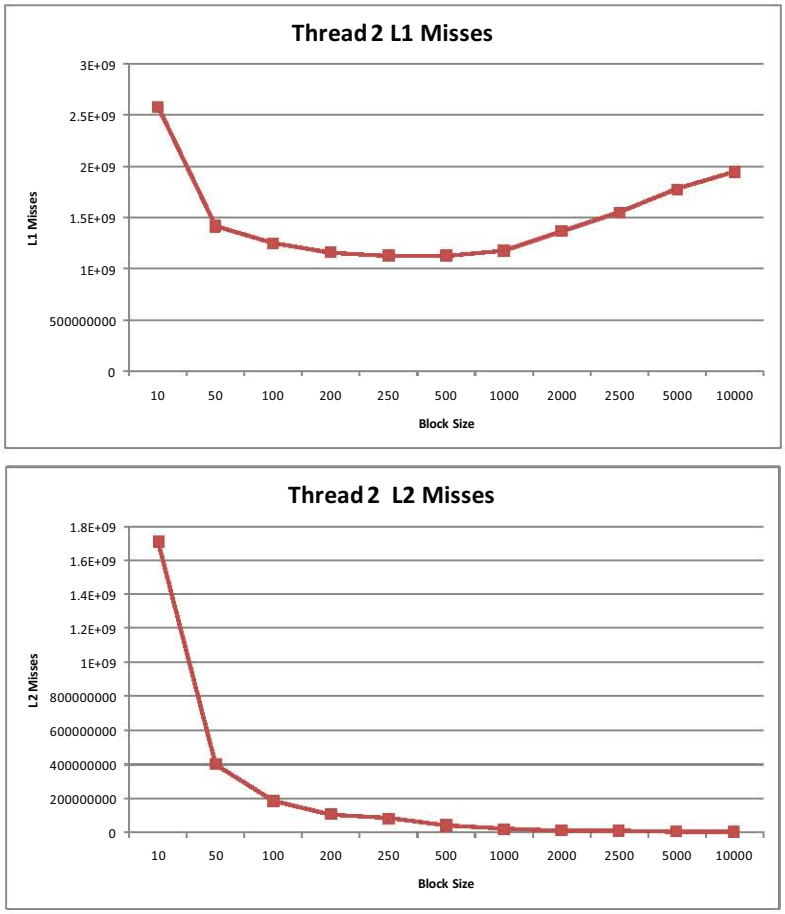

Figure 5.5: Thread 2 L1, L2 Misses on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.5

Table 5.7: Thread 1, Thread 2 CPU Cycles on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity

Block size	Running time of classic parallel algorithm	Thread 1 Cpu Cycles	Thread 2 Cpu Cycles	Total Cycles
10	211.366	7.41E+10	6.32E+10	1.37E+11
50	124.374	4.79E+10	4.39E+10	9.18E+10
100	111.646	4.21E+10	4.53E+10	8.74E+10
200	105.404	4.32E+10	4.29E+10	8.61E+10
250	103.949	4.15E+10	4.18E+10	8.33E+10
500	102.094	3.66E+10	4.65E+10	8.31E+10
1000	104.82	3.43E+10	3.80E+10	7.23E+10
2000	107.081	3.11E+10	4.42E+10	7.53E+10
2500	109.383	2.53E+10	4.54E+10	7.07E+10
5000	120.989	4.93E+10	4.23E+10	9.16E+10
10000	144.558	2.96E+10	4.91E+10	7.87E+10

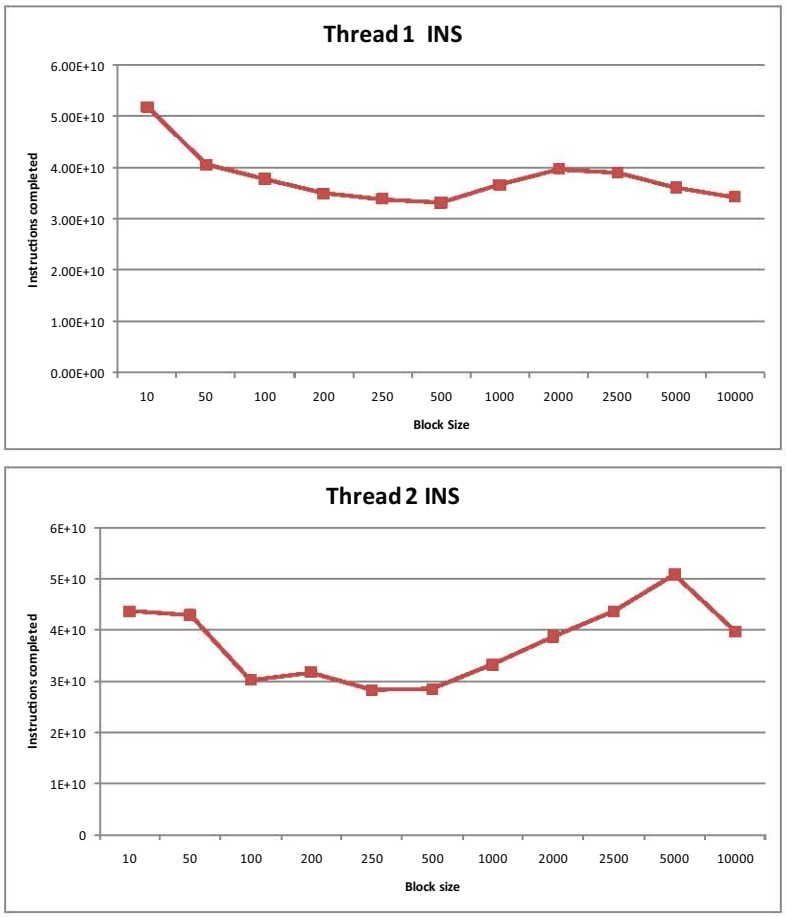

Figure 5.6: Thread 1, Thread 2 Instructions completed on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.6

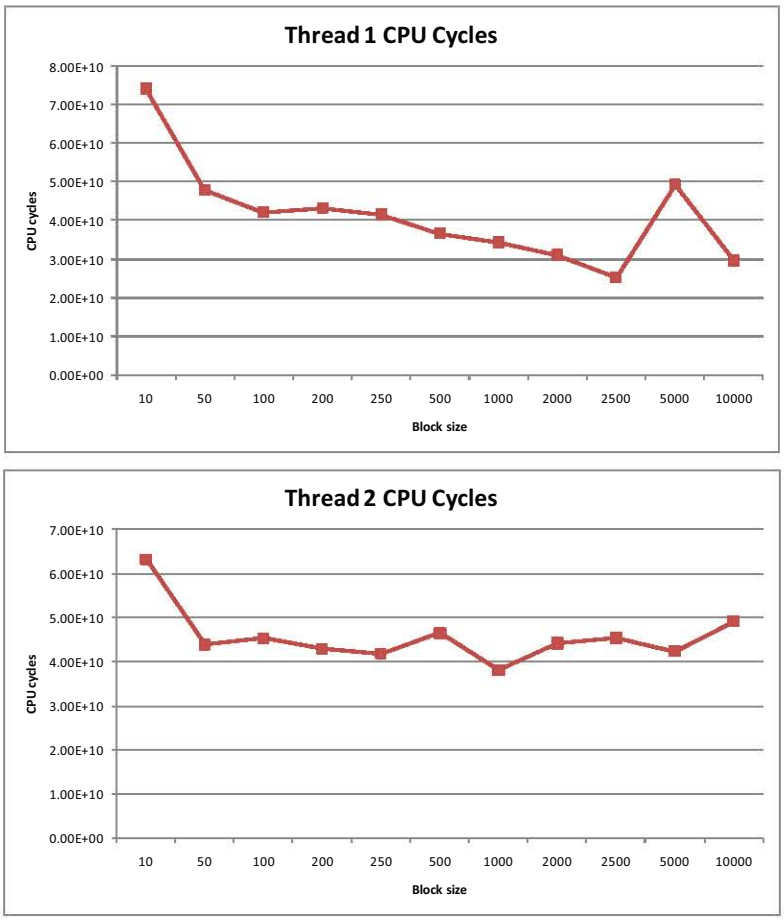

Figure 5.7: Thread 1, Thread 2 CPU Cycles on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.7

Table 5.8: Total L1, L2 misses, CPU cycles and Instruction completed including all Threads on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity

ι size	Algorithm running Time for l1,l2, Instr in sec	Algorithm Running time for CPU Cycles in sec	Total L1 Misses	Total L2 Misses :	Total Instr completed	Total Cpu Cycles
10	208.303	211.366	5.50E+09	2288000000	9.55E+10	1.37E+11
50	121.808	124.374	2.89E+09	588000000	8.35E+10	9.18E+10
100	109.507	111.646	2.53E+09	308000000	6.80E+10	8.74E+10
200	103.371	105.404	2.29E+09	159185418	6.67E+10	8.61E+10
250	101.887	103.949	2.19E+09	128752874	6.21E+10	8.33E+10
500	100.16	102.094	2.11E+09	67506750	6.16E+10	8.31E+10
1000	101.038	104.82	2.03E+09	35543554	6.99E+10	7.23E+10
2000	105.048	107.081	1.98E+09	19371937	7.84E+10	7.53E+10
2500	107.449	109.383	1.97E+09	16101610	8.27E+10	7.07E+10
5000	118.939	120.989	1.93E+09	9550955	8.69E+10	9.16E+10
10000	141.828	144.558	1.98E+09	6290629	7.4E+10	7.87E+10

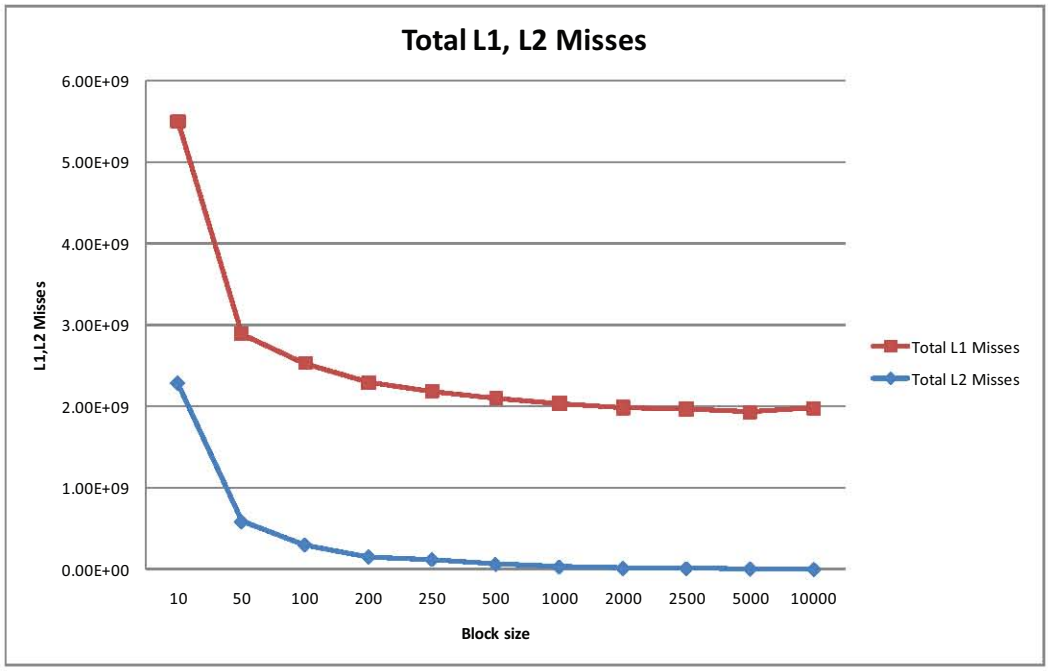

Figure 5.8: Total L1, L2 misses including all Threads on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.8

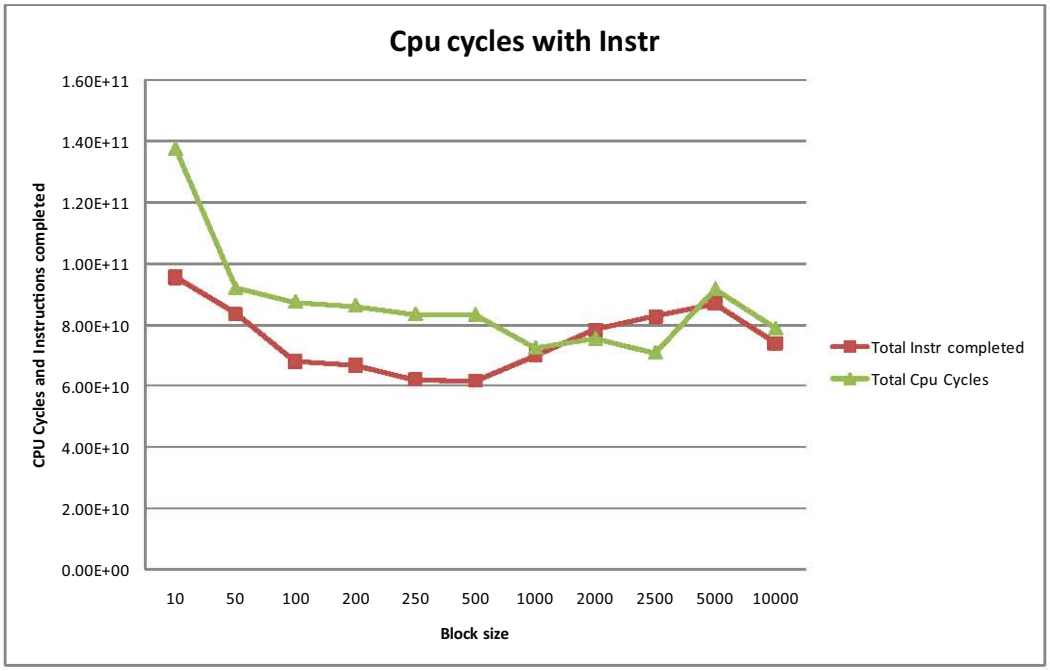

Figure 5.9: Total CPU cycles and Instruction completed including all Threads on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.8

Table 5.9: Total L1, L2 Miss Rates per 1000 ins for all Threads on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity

	Total Instructions Completed	Total L1	Total L2	L1 Miss Rate per INStr	L1 Miss Rates per 1000 INStr	L2 Miss Rate per INStr	L2 Miss Rates per 1000 INStr
10	95500000000	5500000000	2288000000	0.057591623	57.59162	0.023958	23.95812
50	83500000000	2890000000	588000000	0.034610778	34.61078	0.007042	7.041916
100	68000000000	2530000000	308000000	0.037205882	37.20588	0.004529	4.529412
200	66700000000	2290000000	159185418	0.034332834	34.33283	0.002387	2.386588
250	62100000000	2190000000	128752874	0.0352657	35.2657	0.002073	2.073315
500	61600000000	2105000000	67506750	0.034172078	34.17208	0.001096	1.095889
1000	69900000000	2034000000	35543554	0.029098712	29.09871	0.000508	0.508491
2000	78400000000	1984000000	19371937	0.025306122	25.30612	0.000247	0.247091
2500	82700000000	1970000000	16101610	0.02382104	23.82104	0.000195	0.194699
5000	86900000000	1934000000	9550955	0.022255466	22.25547	0.00011	0.109907
0000	74000000000	1981793179	6290629	0.026780989	26.78099	8.5E-05	0.085009

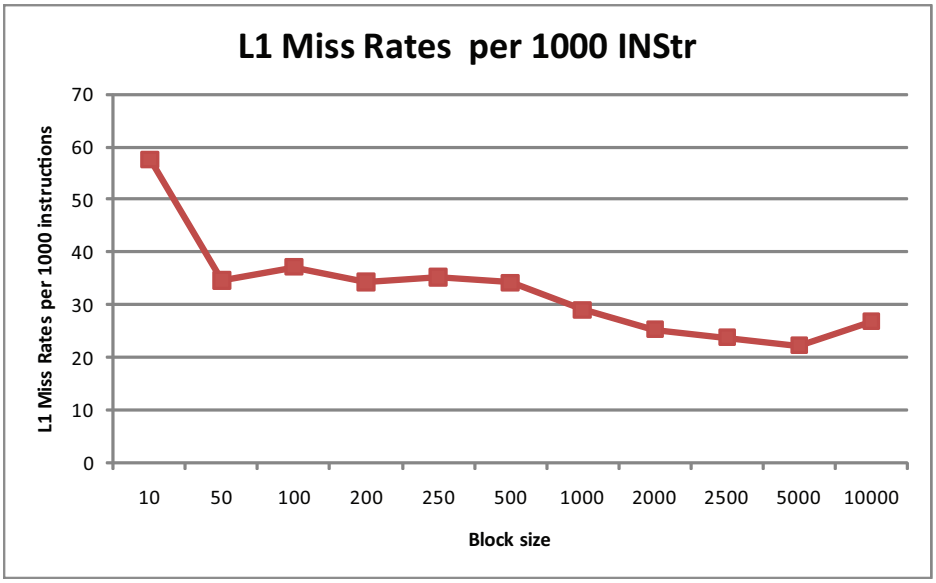

Figure 5.10: Total L1 Miss Rates per 1000 ins for all Threads on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.9

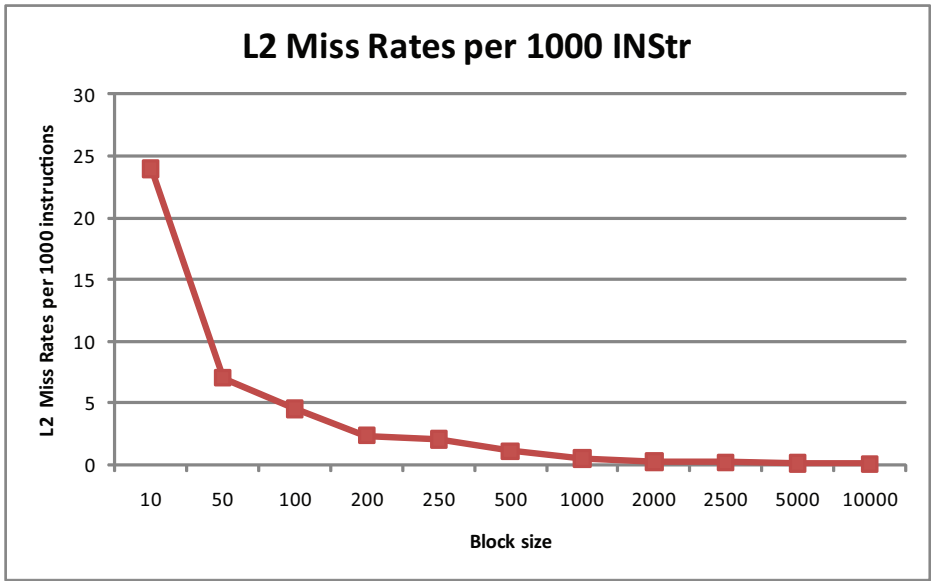

Figure 5.11: Total L2 Miss Rates per 1000 ins for all Threads on *Core2* with Classic parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.9

Morales parallel algorithm analysis for Core2 with the data.dat (10000 x10000) dataset using 2 threads:

Figures. 5.12 to 5.20 show the L1,L2 cache misses, CPU cycles and Instructions completed for the `classic` parallel algorithm. A breakdown of the results for each individual thread are shown as well. The `morales` parallel algorithm had been implemented using pthreads. 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 they are three threads running. Similarly for 4 threads with the *Quad* they are 5 threads in total running and with the 8-core including the main thread they are 9 threads in total. So the data for three threads are shown in the figures 5.12 to 5.20 for the `morales` parallel algorithm for the *Core2* with 2 threads. Thread 1 is the main thread and Thread 2 and Thread 3 are the worker threads.

Table 5.10: Thread 1 L1, L2 Misses on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity

Block Size	L1 Misses	L2 Misses	Running time of Morales parallel algorithm
10	1320132	10001	4.44827
50	1330133	10001	1.53626
100	1350135	10001	1.3119
200	1330133	10001	1.22364
250	1310131	10001	1.20865
500	1320132	10001	1.16916
1000	1280128	10001	1.14916
2000	1340134	10001	1.14085
2500	1310131	10001	1.13936
5000	1300130	10001	1.178
10000	1300130	10001	2.07549

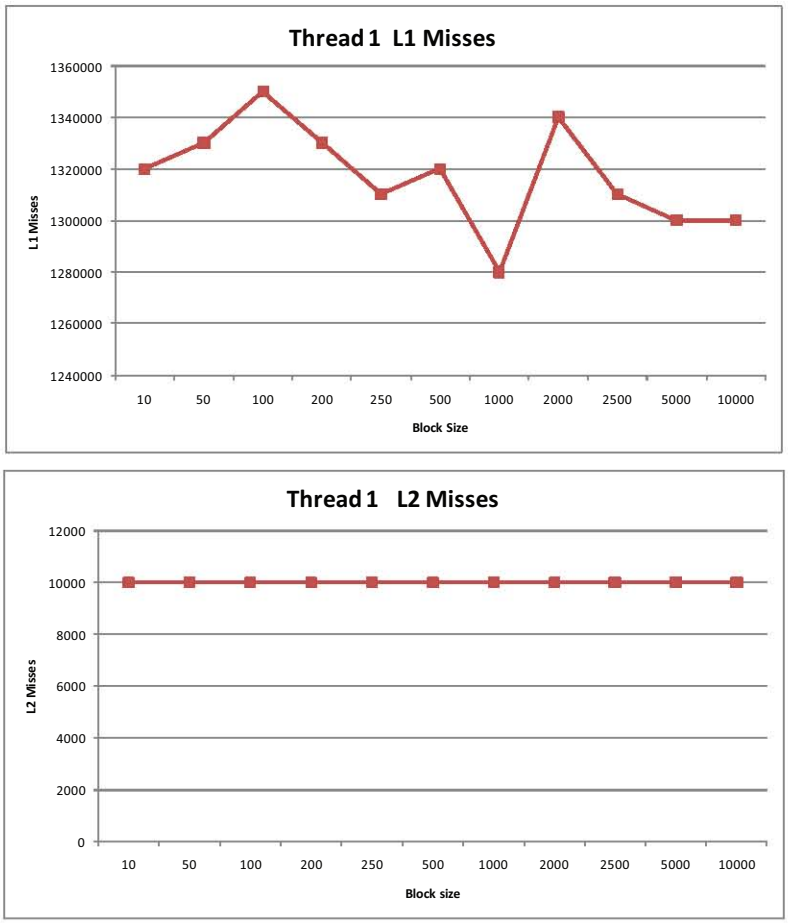

Figure 5.12: Thread 1 L1, L2 Misses on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.10

Table 5.11: Thread 2 L1, L2 Misses on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity

Block Size	L1 Misses	L2 Misses
10	48374837	3070307
50	14501450	3130313
100	11911191	3130313
200	10861086	3130313
250	10671067	3130313
500	10151015	3130313
1000	9770977	3130313
2000	9670967	3130313
2500	9710971	3130313
5000	9860986	3130313
10000	9820982	3130313

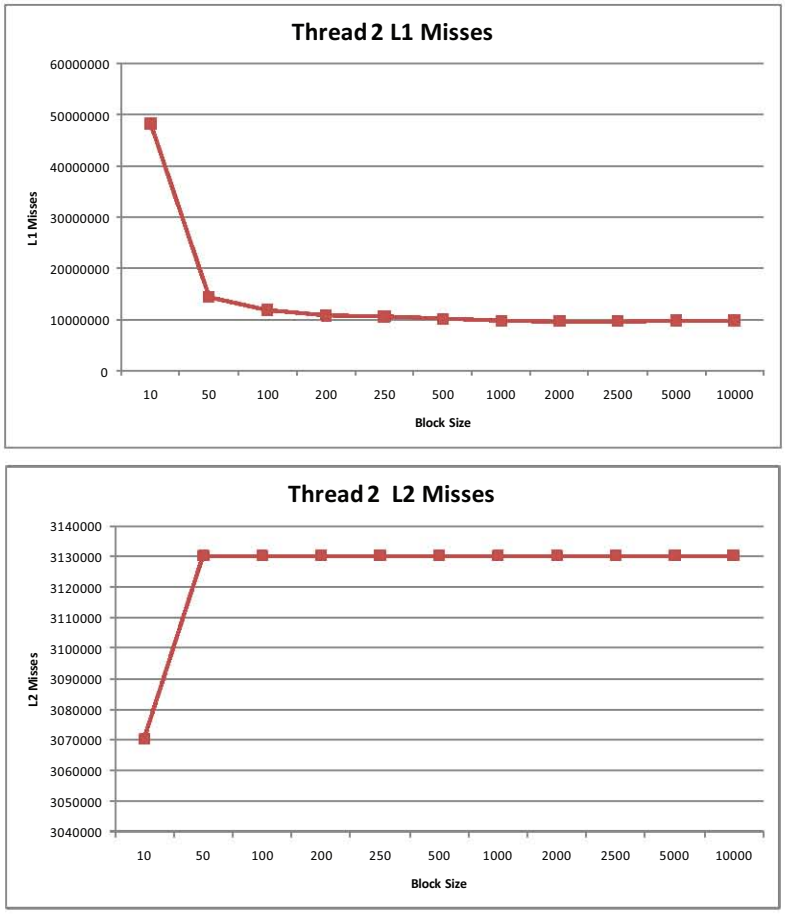

Figure 5.13: Thread 2 L1, L2 Misses on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.11

Table 5.12: Thread 3 L1, L2 Misses on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity

Block Size	L1 Misses	L2 Misses
10	49774977	3190319
50	14581458	3140314
100	11941194	3140314
200	10861086	3130313
250	10671067	3140314
500	10171017	3130313
1000	9800980	3140314
2000	9690969	3130313
2500	9740974	3130313
5000	9900990	3130313
10000	9870987	3130313

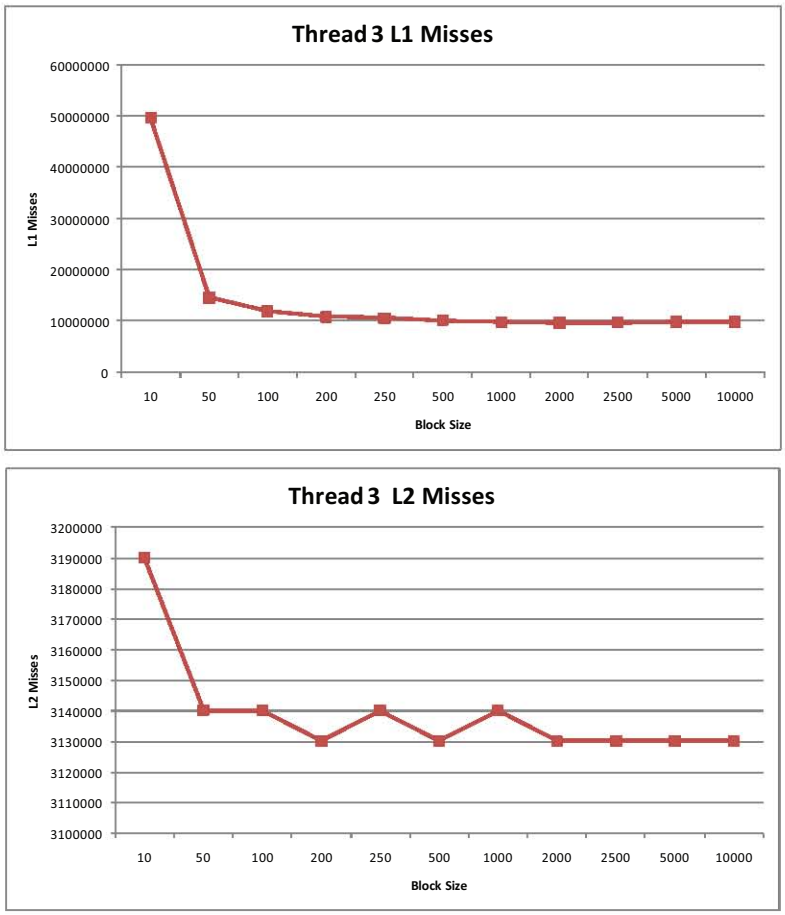

Figure 5.14: Thread 3 L1, L2 Misses on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.12

Table 5.13: Thread 1, Thread 2, Thread 3 Instructions completed on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity

Block Size	Thread 1 INS	Thread 2 INS	Thread 3 INS	Total Instructions Completed
10	1530000000	5.48E+09	5490000000	1.25E+10
50	1530000000	3.92E+09	3920000000	9370000000
100	1530000000	3.73E+09	3730000000	8990000000
200	1530000000	3.63E+09	3630000000	8790000000
250	1530000000	3.61E+09	3610000000	8750000000
500	1530000000	3.58E+09	3580000000	8690000000
1000	1530000000	3.56E+09	3560000000	8650000000
2000	1530000000	3.55E+09	3550000000	8630000000
2500	1530000000	3.55E+09	3550000000	8630000000
5000	1530000000	3.54E+09	3540000000	8610000000
10000	1530000000	3.54E+09	3540000000	8610000000

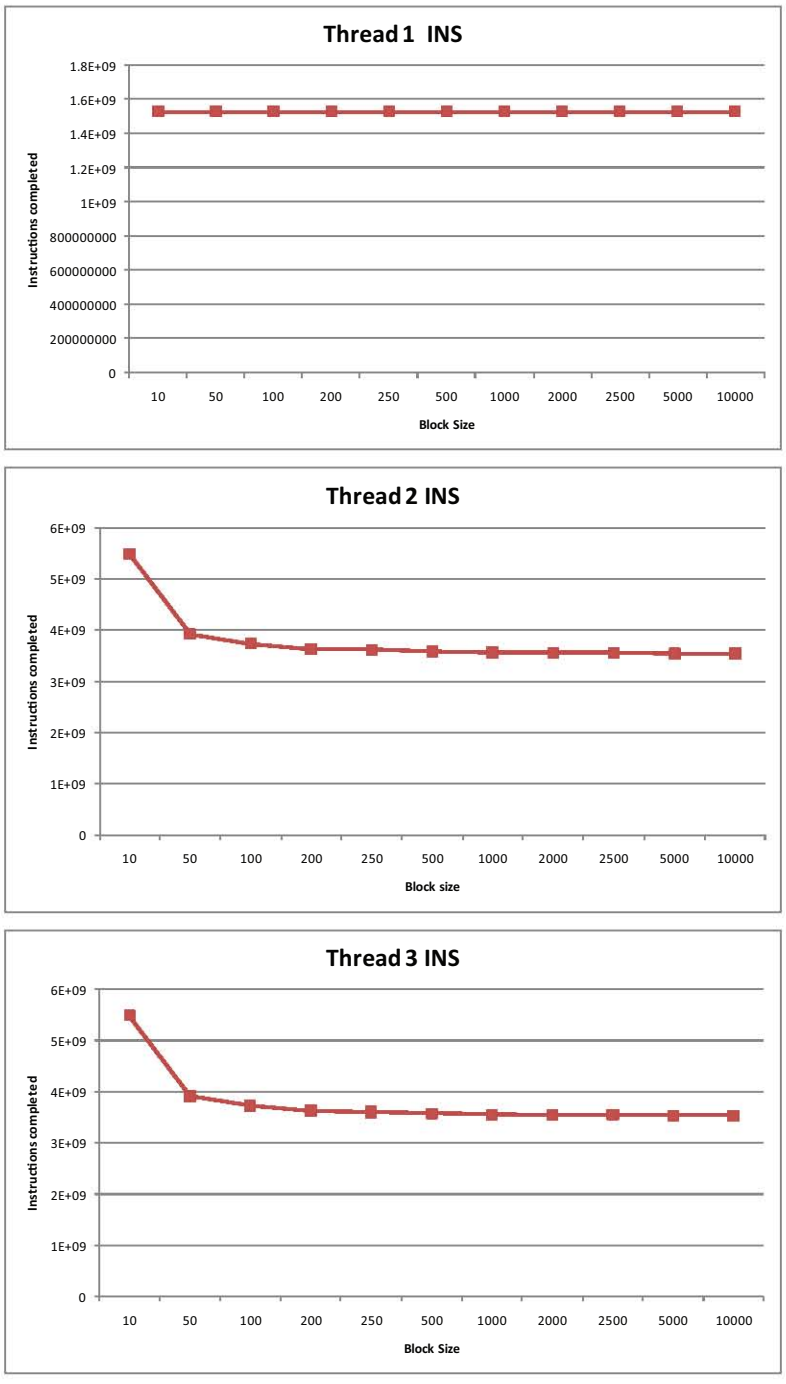

Figure 5.15: Thread 1, Thread 2, Thread 3 Instructions completed on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.13

Table 5.14: Thread 1, Thread 2, Thread 3 CPU Cycles on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity

Block Size	Running time of Morales parallel algorithm	Thread 1 Cpu Cycles	Thread 2 Cpu Cycles	Thread 3 Cpu Cycles	Total CPU Cycles
10	4.63877	1.23E+09	6900000000	6900000000	1.5E+10
50	1.56791	9.34E+08	2930000000	2940000000	6.8E+09
100	1.33757	9.82E+08	2510000000	2520000000	6.01E+09
200	1.24879	1.12E+09	2310000000	2320000000	5.75E+09
250	1.23046	1.05E+09	2270000000	2270000000	5.59E+09
500	1.1882	1.34E+09	2190000000	2190000000	5.72E+09
1000	1.16864	1.22E+09	2150000000	2150000000	5.52E+09
2000	1.16424	1.14E+09	2130000000	2130000000	5.4E+09
2500	1.15788	1.29E+09	2130000000	2130000000	5.55E+09
5000	1.20013	1.26E+09	2130000000	2130000000	5.52E+09
10000	2.10711	1.14E+09	2120000000	2120000000	5.38E+09

Table 5.15: Total L1, L2 misses, CPU cycles and Instruction completed including all Threads on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity

Block size	Total L1 Misses	Total L2 Misses	Total Instructions Completed	Total CPU Cycles	Algorithm Running Times for INS and Cycles in Sec	Algorithm Running Time for L1 , L2 in Sec
10	99469946	6270627	12500000000	15030000000	4.63877	4.44827
50	30413041	6280628	9370000000	6804000000	1.56791	1.53626
100	25202520	6280628	8990000000	6012000000	1.33757	1.3119
200	23052305	6270627	8790000000	5750000000	1.24879	1.22364
250	22652265	6280628	8750000000	5590000000	1.23046	1.20865
500	21642164	6270627	8690000000	5720000000	1.1882	1.16916
1000	20852085	6280628	8650000000	5520000000	1.16864	1.14916
2000	20702070	6270627	8630000000	5400000000	1.16424	1.14085
2500	20762076	6270627	8630000000	5550000000	1.15788	1.13936
5000	21062106	6270627	8610000000	5520000000	1.20013	1.178
10000	20992099	6270627	8610000000	5380000000	2.10711	2.07549

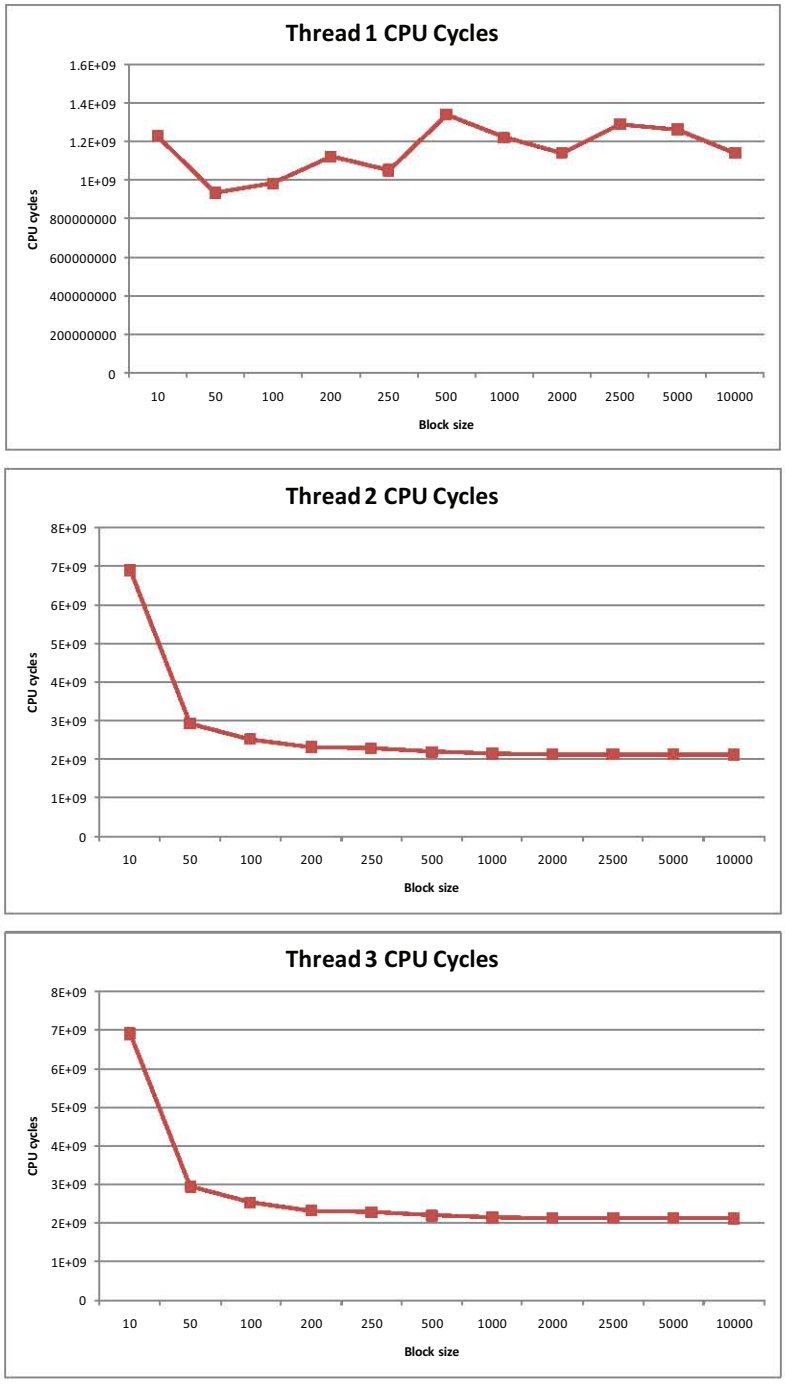

Figure 5.16: Thread 1, Thread 2, Thread 3 CPU Cycles on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity. Graph from Table 5.14

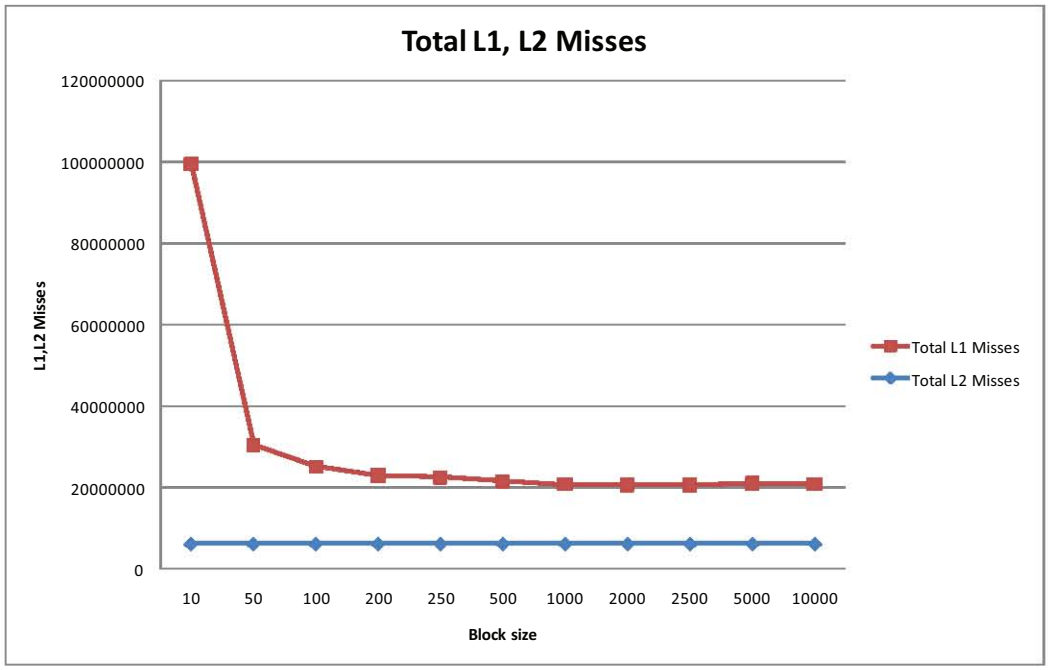

Figure 5.17: Total L1, L2 misses including all Threads on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.15

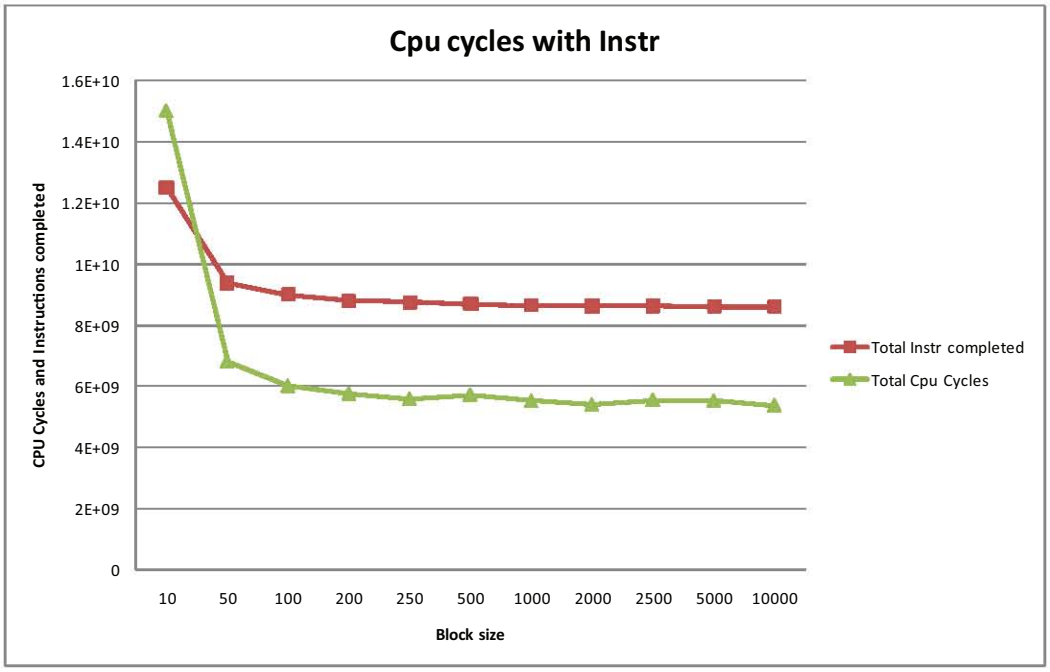

Figure 5.18: Total CPU cycles and Instruction completed including all Threads on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.15

Table 5.16: Total L1, L2 Miss Rates per 1000 ins for all Threads on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity

Block Size	Total Instructions Completed	Total L1 Misses	Total L2 Misses	L1 Miss Rate per INStr	L1 Miss Rates per 1000 INStr	L2 Miss Rate per INStr	L2 Miss Rates per 1000 INStr
10	12500000000	99469946	6270627	0.007957596	7.95759568	0.00050165	0.50165016
50	9370000000	30413041	6280628	0.003245789	3.245788794	0.000670291	0.670291142
100	8990000000	25202520	6280628	0.002803395	2.803394883	0.000698624	0.698623804
200	8790000000	23052305	6270627	0.00262256	2.622560296	0.000713382	0.713381911
250	8750000000	22652265	6280628	0.00258883	2.588830286	0.000717786	0.717786057
500	8690000000	21642164	6270627	0.002490468	2.490467664	0.000721591	0.721591139
1000	8650000000	20852085	6280628	0.002410646	2.410645665	0.000726084	0.726084162
2000	8630000000	20702070	6270627	0.002398849	2.398849363	0.000726608	0.726607995
2500	8630000000	20762076	6270627	0.002405803	2.405802549	0.000726608	0.726607995
5000	8610000000	21062106	6270627	0.002446238	2.446237631	0.000728296	0.728295819
10000	8610000000	20992099	6270627	0.002438107	2.438106736	0.000728296	0.728295819

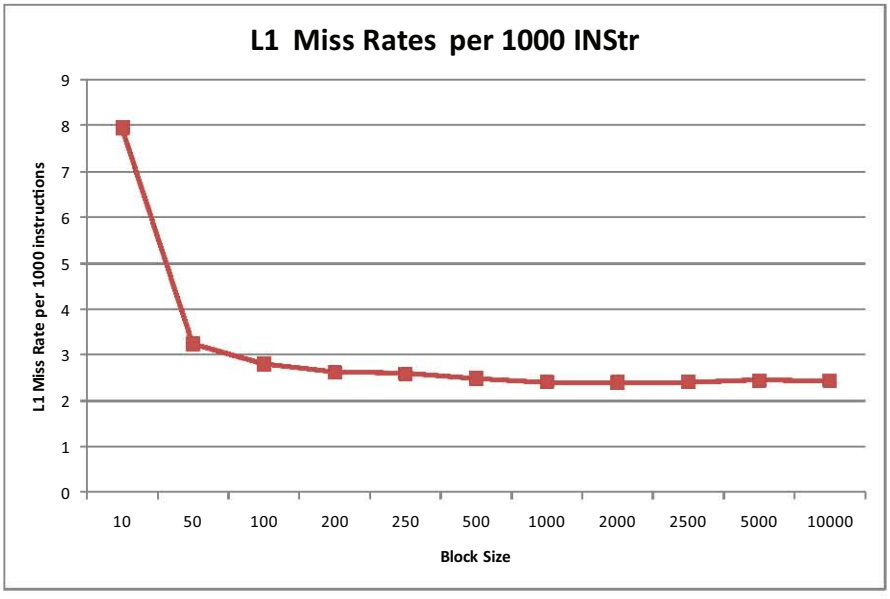

Figure 5.19: Total L1 Miss Rates per 1000 ins for all Threads on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.16

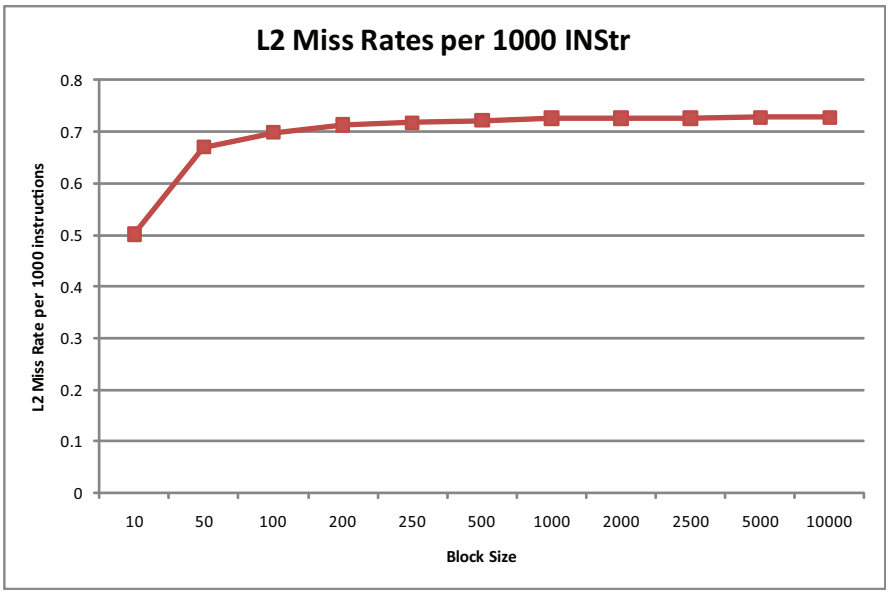

Figure 5.20: Total L2 Miss Rates per 1000 ins for all Threads on *Core2* with Morales parallel algorithm using data.dat : 10000 items x 10000 capacity, Graph from Table 5.16

Summary of results for Core2 with the data.dat (10000 x10000) dataset using 2 threads:

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. The results of these experiments are summarized in fig. 5.2 and fig. 5.3. These figures show performance of `classic` and `morales` on *Core2*, as the block sizes are varied. One observes a clear performance trend for both `classic` and `morales`. 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 $5K$.

The poor performance for smaller block sizes is speculated to be due to a result of poor granularity. When block sizes are ≤ 48, concurrent threads are not assigned enough computation to offset the overhead of thread creation and synchronization.

In the overall L1,L2 Cache misses of the `classic` algorithm in table 5.8,fig. 5.8, table 5.9, fig. 5.10 and fig. 5.11 one can see that both the L1 , L2 misses decrease and became fairly constant after block size 1000. The CPU cycles for the classic algorithm in fig. 5.9 and table 5.8 constantly decrease till they hit block size 1000 and then start to increase after block size 2500. The instructions completed from fig. 5.9 and table 5.8 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 the CPU cycles and instructions completed when the block size is less than 48. This goes hand in hand with the values from fig. 5.2. Looking at the individual thread figures 5.4 to 5.7 a similar pattern is seen.

Looking at the overall L1 cache misses of the `morales` algorithm in table 5.15 and fig. 5.17 one see that they start high and decrease after block size 50k and became somewhat constant. The L2 cache misses in these figures remains fairly constant and unchanged. The CPU cycles and Instructions completed of the `morales` algorithm in table 5.15 and fig. 5.18 also follow a similar trend like the L1 misses. So they are higher CPU cycles, L1 misses and Instructions completed when the block size is less than 48. These results follow a similar trend like that in fig. 5.3 when the block size is less than 48. Looking at the individual thread figures 5.12 to 5.16 a similar pattern is seen.

On the other hand, when the block size to is greater than 5K, the speed up is decreased. In the `classic` parallel algorithm when the block size is greater than 5K, one sees that both the CPU cycles from fig. 5.9 and table 5.8 and the Instructions completed increase. The L1, L2 misses (table 5.8,fig. 5.8, table 5.9, fig. 5.10 and fig. 5.11) stay fairly constant beyond this point. So one reason for the lower performance of the classic parallel algorithm beyond this point of 5K can be explained with the increase in the CPU cycles and Instructions completed. Also in fig. 5.5 the L1 misses of Thread 2 go up after a block size of 500. The instructions completed of Thread 2 in fig. 5.6 also go up after 500. With Thread 1 in fig. 5.6 the instructions completed go up slightly after 500. In CPU cycles Thread 2 in fig. 5.7 goes up slightly after 1000 while Thread 1 gets a sharp increase after the block size of 2500. The maximum speed up with the parallel classic was around the block size of 500. After that the speed up drops and these individual thread figures show that. The individual thread figures show similar results to those of the overall total CPU

cycles and Instructions completed of the algorithm. But they also give an interesting insight with the L1,L2 misses. Looking at the individual threads , it is seen that although the L1,L2 misses in total are dropping but with Thread 2 the L1 misses increase after the block size of 500. So apart from the increase in CPU cycles and the Instructions completed of the classic parallel algorithm, This could be another reason for the decrease in its speed up.

Looking at the L1,L2 cache misses,CPU cycles and Instructions completed of the `morales` parallel algorithm from table 5.15, and figures 5.17 and 5.18, one sees that after block size 5K everything is mostly constant. This does not tell much as to why after 5K the speed up is decreased with the `morales` algorithm, unlike the `classic algorithm` where the CPU cycles and Instructions completed did show an increase in activity beyond this point, along with Thread 2's L1 cache misses. In the `morales` parallel algorithm Thread 1's L1 misses in fig 5.12 increase after the block size of 1000 and Thread 1's CPU cycles in fig 5.16 are steadily increasing. The Threads 2 and 3 either become constant at some point or have decreasing numbers.

So it is possible that apart from Thread 1, something else is happening with Thread 2 and 3 in the `morales` parallel algorithm beyond this point. It is possible that the L1,L2 misses could be resulting in page faults increasing the running time of the algorithm. So although the misses are constant , after the 5K mark they could be leading to page faults instead.

5.3.1.2 Quad with 4 threads:

Table 5.17: Impact of block size on performance of parallel `classic` on *Quad* using data.dat : 10000 items x 10000 capacity

Quad : 4 threads data.dat : 10000 items x 10000 capacity			
Classic			
Block size	**Parallel Running time in Seconds**	**Sequential running time in seconds**	**Speed up**
		174.952	
2	(very long)		
10	556.253		0.314519
50	163.192		1.072062
100	108.679		1.609805
200	80.0991		2.184194
250	72.75		2.404838
500	61.9131		2.825767
1000	57.1973		3.058746
2000	58.0052		3.016143
2500	59.7216		2.929459
5000	70.6995		2.474586
10000	89.6722		1.951017

Running times and Speed up with varying block sizes

Figs. 5.21 and 5.22 present performance results for varying block sizes on *Quad*. One thing to note that both graphs of the respective algorithms achieve higher speed ups than the *Core2*. An identical performance pattern is observed on this platform with the `morales` parallel algorithm compared to the *Core2* from before. Any block size smaller than 48 or larger than $5K$ turns out to be a poor choice. With a block size of less than 48 the `classic` parallel algorithm shows similar results to the *Core2* as well. When the block size is larger than 5K the `classic` parallel algorithm still shows speed up, even with the largest block size of 10000. However the pattern of the speed up overall is similar to the one before with the *Core2*.

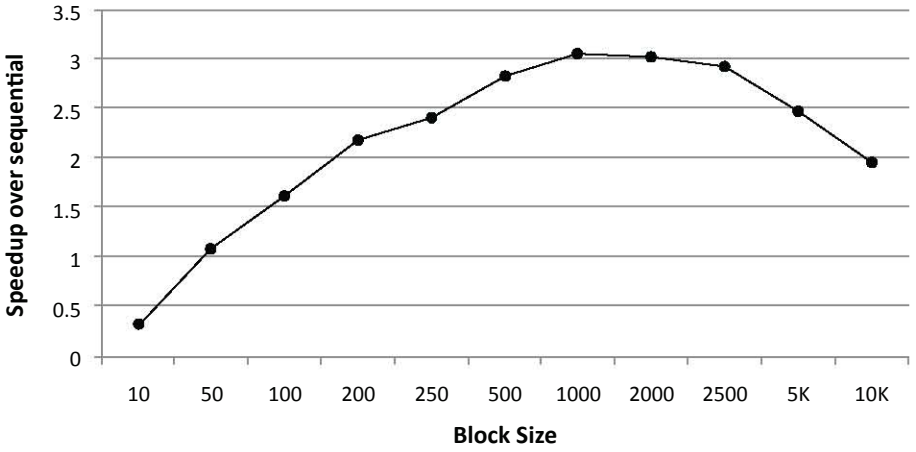

Figure 5.21: Impact of block size on performance of parallel `classic` on *Quad* using data.dat : 10000 items x 10000 capacity, Graph from Table 5.17

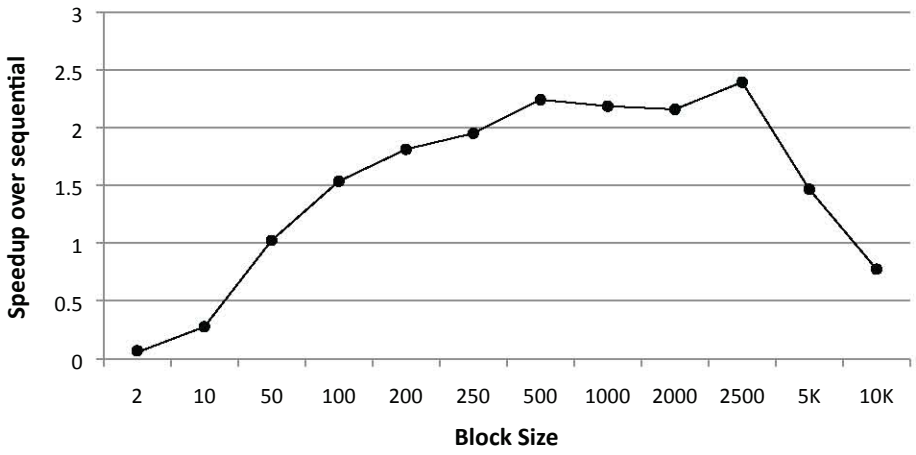

Figure 5.22: Impact of block size on performance of parallel `morales` on *Quad* using data.dat : 10000 items x 10000 capacity, Graph from Table 5.18

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. The range of good tile sizes are likely to be different for architectures that have cache configurations that are significantly different from *Core2* or *Quad*.

Table 5.18: Impact of block size on performance of parallel `morales` on *Quad* using data.dat : 10000 items x 10000 capacity

Quad : 4 threads **data.dat : 10000 items x 10000 capacity**			
Morales			
Block size	**Parallel Running time in Seconds**	**Sequential running time in seconds**	**Speed up**
		1.42791	
2	23.8235		0.059937
10	5.19119		0.275064
50	1.40146		1.018873
100	0.929053		1.536952
200	0.786962		1.814459
250	0.735538		1.941314
500	0.635935		2.245371
1000	0.655799		2.177359
2000	0.662451		2.155495
2500	0.595529		2.397717
5000	0.974834		1.464772
10000	1.85246		0.770818

5.3.1.3 8-core with 8 threads:

Table 5.19: Impact of block size on performance of parallel `classic` on *8-core* using data.dat : 10000 items x 10000 capacity

8-core : 8 threads **data.dat : 10000 items x 10000 capacity**			
Classic			
Block size	**Parallel Running time in Seconds**	**Sequential running time in seconds**	**Speed up**
		168.472	
10	89.3494		1.885541
50	37.5553		4.485971
100	30.5213		5.519817
200	25.5953		6.582146
250	25.5377		6.596992
500	24.3337		6.923403
1000	24.7436		6.80871
2000	25.831		6.522086
2500	26.016		6.475707
5000	29.9237		5.630052
10000	38.7219		4.35082

Running times and Speed up with varying block sizes

Figs. 5.23 and 5.24 present performance results for varying block sizes on *8-core*.One thing to note that both graphs of the respective algorithms achieve higher speed ups than the *Quad*.In the `classic` algorithm even with smaller block sizes there is still speed up. The pattern of the speed up of the algorithm is similar to the ones (of the classic algorithm) seen before. In the `morales` algorithm there is no speed up when the block size is less than 10. The behaviour of having block sizes greater than 5K is the same. The speed up is lost when the block sizes are less than 10 and greater than 5K. Since the speed up overall is greater than the *Quad* with similar behaviours the graph is more steeper than the ones seen from before. In all

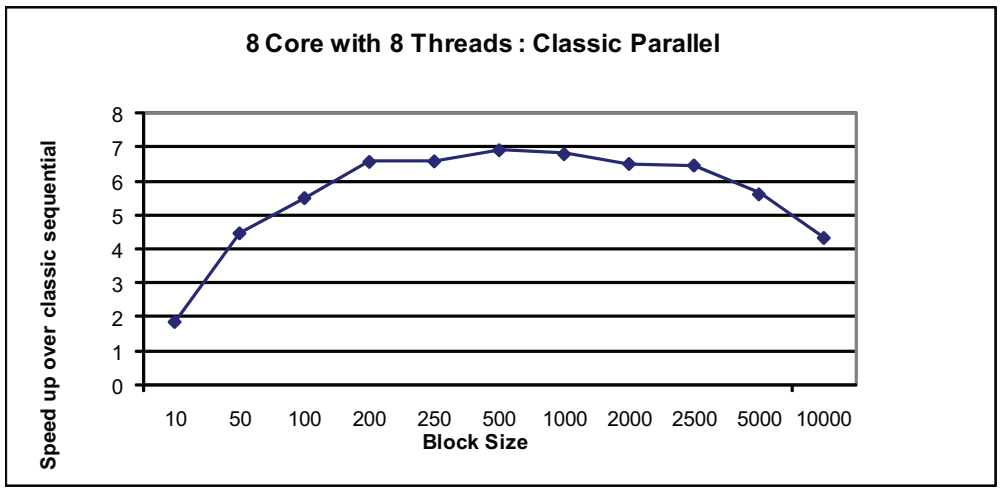

Figure 5.23: Impact of block size on performance of parallel `classic` on *8-core* using data.dat : 10000 items x 10000 capacity, Graph from Table 5.19

of these graphs of both the algorithms with the different machines a similar bell

type shape block size pattern is observed.

Table 5.20: Impact of block size on performance of parallel `morales` on *8-core* using data.dat : 10000 items x 10000 capacity

8-core : 8 threads **data.dat : 10000 items x 10000 capacity**			
Morales			
Block size	**Parallel Running time in Seconds**	**Sequential running time in seconds**	**Speed up**
		1.2407	
2	9.57458		0.129583
10	2.25421		0.550392
50	0.632238		1.962394
100	0.378743		3.275836
200	0.281235		4.411613
250	0.252115		4.921167
500	0.270831		4.581086
1000	0.2825		4.391858
2000	0.645026		1.923488
2500	0.697792		1.778037
5000	1.2279		1.010424
10000	1.89505		0.654706

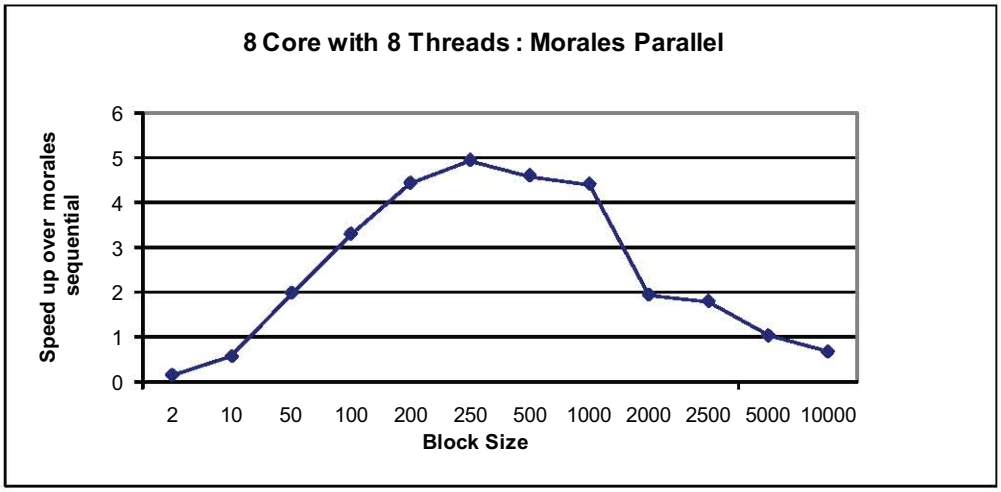

Figure 5.24: Impact of block size on performance of parallel `morales` on *8-core* using data.dat : 10000 items x 10000 capacity, Graph from Table 5.20

5.4 Impact of Data Set Size and Range of Values

The following figures show the running times of both parallel `classic` and `morales` with the variation in block sizes with larger datasets and different values of **R**.

5.4.1 Using the Data set : ds60-30k.dat : 30000 items x 30000 capacity.

One thing to note in this test is that the dataset's **R** value is 30000. The `classic` sequential algorithm overall has a better running time with this dataset. It is better than the smaller data.dat (**R**=100 ,10000 x10000) dataset even though the size of 30000 items x 30000 capacity is larger. One reason for this could be the iterations of the inner while loop (0 ¡ l ¡ $\hat{c}$/weightitem) in the algorithm. When **R**=100 this means that the dataset has smaller values 1:100 and therefore more combinations. When **R**=30000 this means that the values are larger and hence lesser combinations and so lesser iterations of the while loop. This is proved when another dataset (**ds30-300.dat**) is used of the same size 30000 items x 30000 capacity with **R**=300. The fig 5.25 and table 5.21 shows the results. The `classic` sequential and parallel algorithm were run on the 8-core machine with 8 threads.

From the table 5.21 one see the large running times required from the classic sequential to run such a dataset with smaller R values and larger sizes of items and capacity.

The *Core2* machine could not be used for this test because it would run out of memory when running a dataset of this size and the **data.dat** (10000 x10000)

Table 5.21: `Classic` parallel on *8-core* using ds30-300.dat (R:300, 30000 items x 30000 capacity)

8-core : 8 threads ds30-300.dat : 30000 items x 30000 capacity R:300			
Classic			
Block size	**Parallel Running time in Seconds**	**Sequential running time in seconds**	**Speed up over sequential**
		2068	
2500	1333.56		1.550736375
5000	324.15		6.379762456
10000	365.037		5.665179146
15000	402.707		5.135247214
30000	522.072		3.96113946

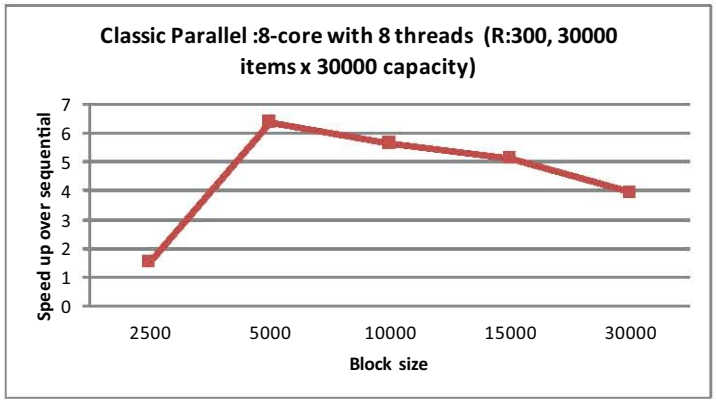

Figure 5.25: `Classic` parallel on *8-core* using ds30-300.dat (R:300, 30000 items x 30000 capacity), Graph from Table 5.21

dataset was among the maximum it could take.

5.4.1.1 Overall Speed up of Quad and 8-core compared

Table 5.22 and Fig. 5.26 show speedup obtained over the sequential version, for both `classic` and `morales` for 4 and 8 cores. The data set used was ds60-30k.dat with 30000 item x 30000 capacity and R=30000. This chart reveals that both parallel variants obtain significant speedup over their sequential counterparts. A

Table 5.22: Performance improvement with increasing number of cores with ds60-30k.dat dataset (R:30000, 30000 items x 30000 capacity)

	Parallel Algorithms Speed up over Sequential Counterparts	
#Cores Machines with same #Threads	Classic	Morales
4	2.256168	2.648065
8	5.78876	5.145088

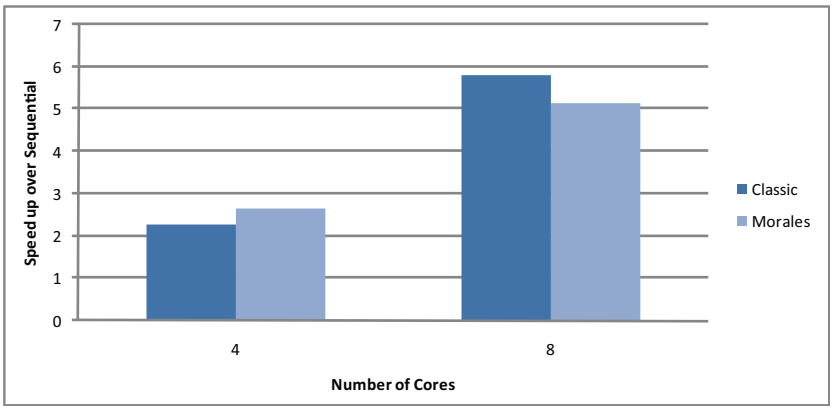

Figure 5.26: Performance improvement with increasing number of cores with ds30-60k.dat dataset (R:30000, 30000 items x 30000 capacity), Graph from Table 5.22

more detailed view of the block size pattern is shown for both the Quad and 8-core in the following sections.

5.4.1.2 Quad with 4 threads:

Running times and Speed up with varying block sizes

In figure 5.27 we see an increasing pattern of speed up of the `classic` parallel algorithm starting from a block size of 1000 up to a block size of 15k. Before the block size 1000 the speed up is not present but starting from 50 the performance does continue to increase. After the block size of 15k the performance decreases but there is still speed up. In the fig 5.28 of the `morales` parallel algorithm we see a more familiar bell shape type pattern of the speed up. The behavior seems similar

Table 5.23: Impact of block size on performance of parallel `classic` on *Quad* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity)

Core 4 Quad : 4 threads ds60-30k.dat : 30000 x 30000			
Classic			
Block size	**Parallel Algrthm Running time in secs**	**Sequential Algrthm Running time in secs**	**Speed up over sequential**
		48.6784	
50	873.953		0.055699
100	447.178		0.108857
200	230.647		0.211052
250	189.382		0.257038
500	104.839		0.464316
1000	61.4949		0.791584
2000	39.6362		1.22813
2500	35.4272		1.37404
5000	26.6018		1.829891
10000	22.5772		2.156087
15000	21.5757		2.256168
30000	24.2907		2.003993

to the one with the data.dat dataset from before. Before the block size 50 there is no speed up and after block size 30k there is no speed up. The performance increases till it reaches 5K and then decreases afterwards.

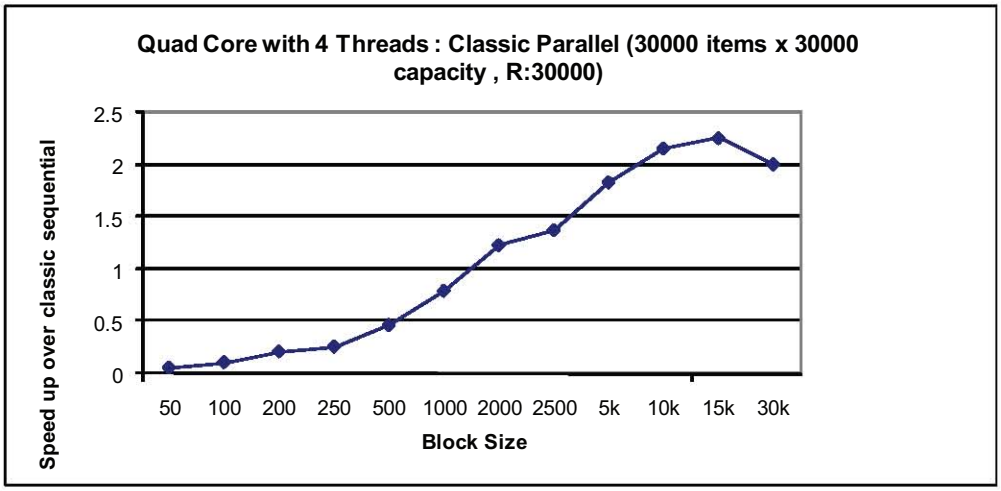

Figure 5.27: Impact of block size on performance of parallel `classic` on *Quad* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity), Graph from Table 5.23

Table 5.24: Impact of block size on performance of parallel `morales` on *Quad* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity)

Core 4 Quad : 4 threads ds60-30k.dat : 30000 x 30000			
Morales			
Block size	**Parallel running time in Secs**	**Sequential running time in secs**	**Speed up over sequential**
		10.5049	
10	44.39		0.23665
50	11.6418		0.902343
100	7.33079		1.432983
200	5.49236		1.912639
250	5.28358		1.988216
500	4.96281		2.116724
1000	5.10009		2.059748
2000	4.91814		2.13595
2500	4.73286		2.219567
5000	3.96701		2.648065
10000	5.46396		1.92258
15000	7.90661		1.328623
30000	12.6438		0.830834

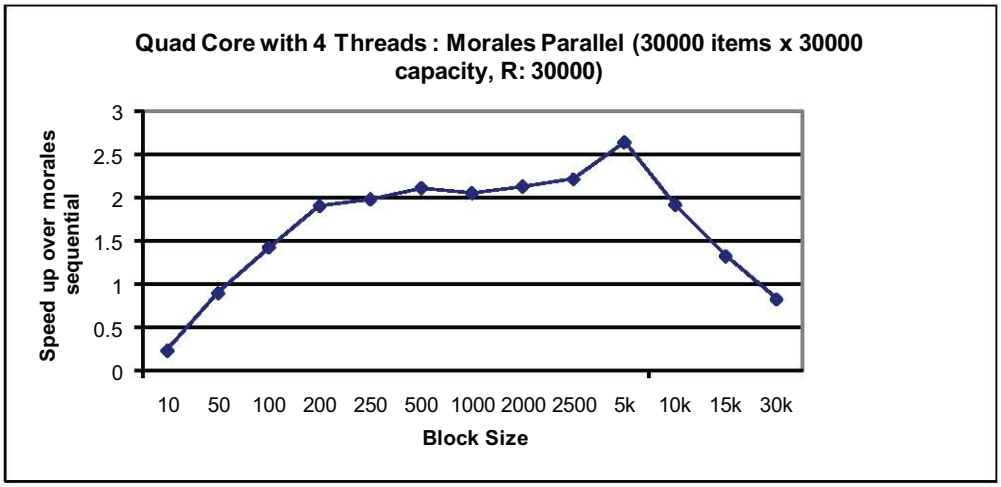

Figure 5.28: Impact of block size on performance of parallel `morales` on *Quad* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity), Graph from Table 5.24

5.4.1.3 8-core with 8 threads:

Table 5.25: Impact of block size on performance of parallel `classic` on *8-core* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity)

8-Core : 8 threads ds60-30k.dat : 30000 x 30000			
Classic			
Block size	Parallel algorithm Running time in secs	Sequential algorithm Running time in secs	Speed up over sequential
		43.6777	
10	290.246		0.150485
50	67.143		0.650518
100	36.073		1.210814
200	21.9986		1.985476
250	20.4762		2.133096
500	13.6401		3.202154
1000	10.5676		4.133171
2000	8.74639		4.993797
2500	8.08234		5.404091
5000	7.54526		5.78876
10000	7.73978		5.643274
15000	8.21835		5.314656
30000	9.97752		4.377611

Running times and Speed up with varying block sizes

In fig. 5.29 we see a similar pattern of the `classic` parallel algorithm to the one seen before with the same dataset on the *Quad*. The graph achieves higher speed up than the *Quad*. It gains speed up after a smaller block size of 100 and continues to have a speed up with the largest block size of 30000. In this case the maximum speed up is reached sooner with a lower block size of 5000 , after which the performance decreases. In fig. 5.30 we again see the bell type shape pattern of the `morales` parallel algorithm. The speed up is a lot more than the *Quad*. The speed up starts from a smaller block size of around 20 and ends after the block size of 15k.

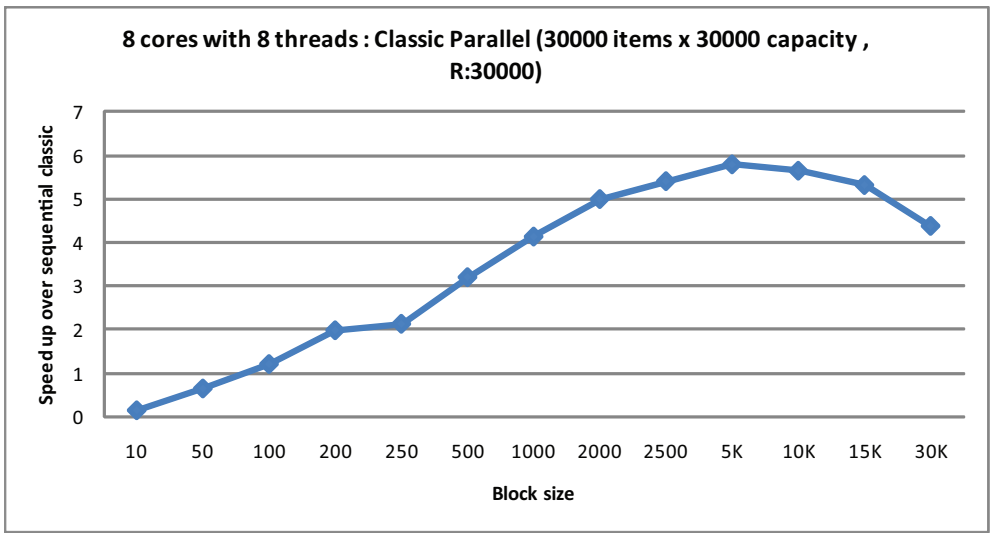

Figure 5.29: Impact of block size on performance of parallel `classic` on *8-core* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity), Graph from Table 5.25

5.4.2 Using the Data set : ds60-30k.dat : 60000 items x 60000 capacity.

The same dataset set ds60-30k.dat was used again in this test with the R:30000. The only change was that the items and capacity were both increased to 60000 items and 60000 total capacity. The *Quad* and *Core2* machines could not be used for this test because both of them would run out of memory.

5.4.2.1 Overall Speed up 8-core

Table 5.27 and Fig. 5.31 show speedup obtained over the sequential version, for both `classic` and `morales` for 8 cores. The data set used was ds60-30k.dat with 60000 item x 60000 capacity and R=30000. This chart reveals that both parallel variants obtain significant speedup over their sequential counterparts. A more detailed view of the block size pattern is shown for the 8-core in the following sections.

Table 5.26: Impact of block size on performance of parallel `morales` on *8-core* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity)

8-Core : 8 threads ds60-30k.dat : 30000 x 30000			
Morales			
Block size	**Parallel algorithm Running time in secs**	**Sequential algorithm Running time in secs**	**Speed up over sequential**
		8.54347	
2	89.1302		0.095854
10	19.548		0.437051
50	5.15533		1.657211
100	3.08486		2.769484
200	2.22153		3.845759
250	2.03502		4.198224
500	1.80226		4.74042
1000	1.71406		4.984347
2000	1.66051		5.145088
2500	1.68942		5.057043
5000	2.58473		3.305363
10000	5.36337		1.592929
15000	7.80193		1.095046
30000	12.0662		0.70805

5.4.2.2 8-core with 8 threads:

Running times and Speed up with varying block sizes

In fig. 5.32 we see a similar pattern of the `classic` parallel algorithm to the one seen before with the same dataset with 30000 items and 30000 capacity. It gains speed up after a smaller block size of 50 and continues to have a speed up with the largest block size of 60000. The maximum speed up is reached with between a block size of 6000 and 10000 , after which the performance decreases. In fig. 5.33 we again see the bell type shape pattern of the `morales` parallel algorithm.The speed up starts from a smaller block size of around 20 and ends after the block size of 30k.

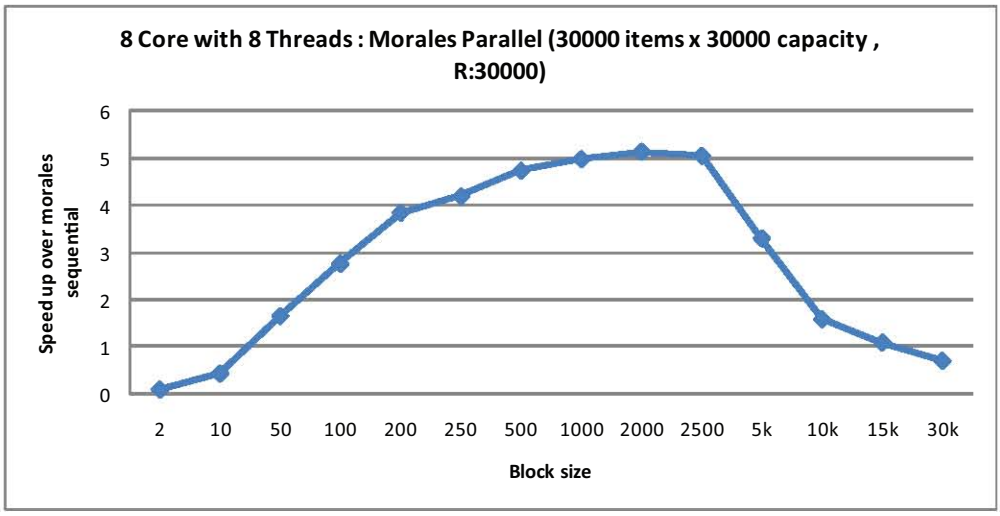

Figure 5.30: Impact of block size on performance of parallel `morales` on *8-core* using ds60-30k.dat (R:30000, 30000 items x 30000 capacity), Graph from Table 5.26

Table 5.27: Performance improvement with increasing number of cores with ds60-30k.dat dataset: 60000 items x 60000 capacity

Data set : ds60-30k.dat : 60000 items x 60000 capacity	Parallel Algorithms Speed up over Sequential Counterparts	
#Cores Machines with same #Threads	Classic	Morales
8	5.924733	5.877096

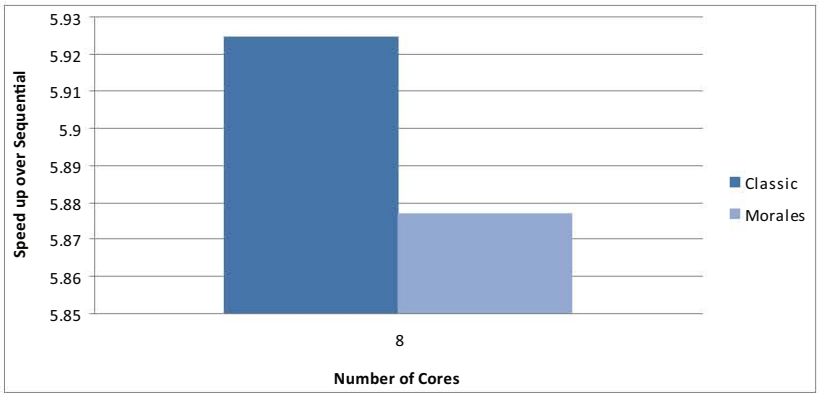

Figure 5.31: Performance improvement with increasing number of cores with ds30-60k.dat dataset: 60000 items x 60000 capacity, Graph from Table 5.27

Table 5.28: Impact of block size on performance of parallel `classic` on *8-core* using ds60-30k.dat (R:30000, 60000 items x 60000 capacity)

8-Core : 8 threads **ds60-30k.dat : 60000 x 60000**			
Classic			
Block Size	**Parallel algorithm Running time in secs**	**Sequential algorithm Running time in secs**	**Speed up over sequential**
		297.549	
10	1248.31		0.238361
50	294.24		1.011246
100	170.853		1.74155
200	114.157		2.606489
250	104.328		2.852053
300	94.5814		3.145957
500	77.9818		3.815621
1000	67.0304		4.439016
2000	58.2024		5.112315
2500	55.7891		5.333461
3000	54.0621		5.503837
5000	51.1151		5.821157
6000	50.2367		5.922941
10000	50.2215		5.924733
12000	50.6974		5.869118
15000	51.7351		5.751395
20000	53.7819		5.532512
30000	58.4627		5.089553
60000	73.5006		4.048253

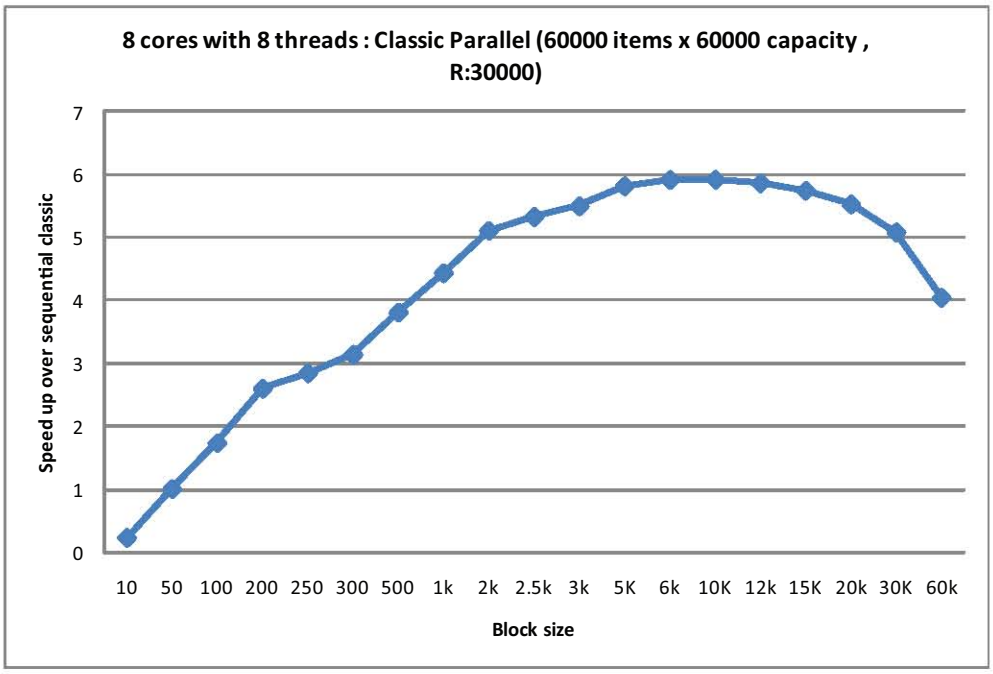

Figure 5.32: Impact of block size on performance of parallel `classic` on *8-core* using ds60-30k.dat (R:30000, 60000 items x 60000 capacity), Graph from Table 5.28

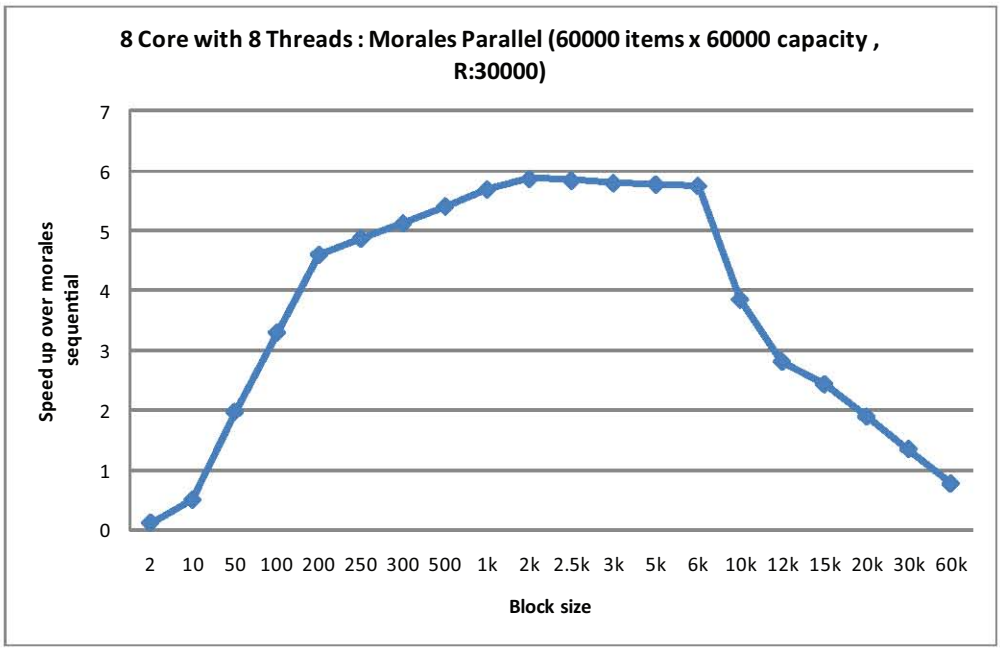

Figure 5.33: Impact of block size on performance of parallel `morales` on *8-core* using ds60-30k.dat (R:30000, 60000 items x 60000 capacity), Graph from Table 5.29

Table 5.29: Impact of block size on performance of parallel `morales` on *8-core* using ds60-30k.dat (R:30000, 60000 items x 60000 capacity)

8-Core : 8 threads **ds60-30k.dat : 60000 x 60000**			
Morales			
Block Size	**Parallel algorithm Running time in secs**	**Sequential algorithm Running time in secs**	**Speed up over sequential**
		39.5468	
2	356.845		0.110823
10	79.4656		0.497659
50	20.0876		1.968717
100	11.9872		3.299086
200	8.59146		4.603036
250	8.11221		4.874972
300	7.7104		5.129021
500	7.30286		5.415248
1000	6.93928		5.698977
2000	6.72897		5.877096
2500	6.76225		5.848172
3000	6.81423		5.803561
5000	6.84038		5.781375
6000	6.86945		5.756909
10000	10.2732		3.849511
12000	14.0776		2.8092
15000	16.2578		2.432482
20000	20.9384		1.888721
30000	29.4532		1.3427
60000	51.8065		0.763356

CHAPTER 6

RELATED WORK

6.1 Parallelization on Multi-core Architecture

[23]Tan et al. discussed a parallel programming algorithm for a multi-core architecture. They presented a scheme to exploit fine grain parallelism and locality of a dynamic programming algorithm with non-uniform dependence on a multi-core architecture. The multi-core 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. [13]Holzmann described a stack slicing algorithm whose application was for multi-core 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. 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. [25]Villa et al. discuss the challenges and design choices involved in parallelizing a breadth-first search algorithm on the Cell Broadband Engine multi-core processor. The Cell BE is meant for high performance

clusters and supercomputers and it's memory hierarchy is explicitly managed at software level. They described how they parallelized the Breadh-first search algorithm and by experimentation proved that their method achieved a high level of performance on the Cell BE processor. [21]Scarpazza, Villa et al. wrote another paper that looked deeper into the parallelization of the Breadth first search algorithm on the Cell BE processor. 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 Previous Work on the Parallelization of the IKP

There is fewer literature on parallelizing the IKP than on parallelizing the 0/1 KP. The related work closest to the one in this paper is in [17]. These authors present multiple strategies for parallelizing the DP recursion 2.2 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 [17] do not consider blocking as an alternative to improve the performance of SPA but they implemented SPA with single dominancy [10] to reduce the high communication cost between processors. The new algorithm is named pipelined algorithm with dominancy (PAD). Only non dominated solutions from a previous row are sent to the next processor to compute 2.2. A future study may consider to combine dominance and the blocking concept proposed in this research to improve performance for `morales` algorithm.

Paper in [17] 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 [2] present a parallel pipelined algorithm for the IKP with time complexity equal to the one in [17], that is, $O(nC/q + n)$ 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 $O(n)$, an improvement to $O(mc)$,the time complexity of usual strategies.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This paper explored the issue of algorithmic choice in the context of the IKP by comparing the performance of two parallel variants on multicore platforms. The 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. Another key finding of this study is that blocking factors have significant impact on performance of each parallel variant. Therefore, to achieve improved performance, it is necessary to select blocking factors through careful analysis. One point to note is that the classic parallel variant was alot simpler to parallelize and was implemented using OpenMP. Transforming a sequential program to a parallel one using OpenMP usually requires minimal code structure changes compared to other methods like pthreads. If one were to compare the ease of parallelizing an algorithm along with the performance gain for a multi-core architecture then the classic parallel variant would be an ideal example. As this demonstrates a generic method in transforming a sequential algorithm with minimal effort and huge performance gains when run on a multi-core system.

The impact of dominance optimization on the sequential IKP was also explored. Dominance significantly reduced the search space of large datasets and improved running times. This improvement in reducing the search space to smaller datasets

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.

BIBLIOGRAPHY

[1] R. Andonov, V. Poirriez, and S. Rajopadhye. Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research*, 123:394–407, 2000.

[2] R. Andonov, F. Raimbault, and P. Quinton. Dynamic programming parallel implementations for the knapsack problem. Technical Report 2037, INRIA Institut National de Recherche en Informatique et en Automatique, France, 1993.

[3] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[4] R. Bellman and S. E. Dreyfus. *Applied Dynamic Programming.* University Press, Princeton, NY, 1962.

[5] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. Quintana-Orti, and R. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[6] G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5:266–288, 1957.

[7] M. Frigo. A fast Fourier transform compiler. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Freeman, San Francisco, NY, 1979.

[9] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem. *Operations Research*, 13:94–120, 1961.

[10] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting stock problem - part ii. *Operations Research*, 11:863–888, 1963.

[11] P. C. Gilmore and R. E. Gomory. Multistage cutting stock problems of two and more dimensions. *Operations Research*, 13:94–120, 1964.

[12] A. Goldman and D. Trystram. An efficient parallel algorithm for solving the knapsack problem on hypercubes. *Journal of Parallel and Distributed Computing*, 64(11):1213–1222, 2004.

[13] G. J. Holzmann. A stack-slicing algorithm for multi-core model checking. *Electron. Notes Theor. Comput. Sci.*, 198(1):3–16, 2008.

[14] A. Kleywegt and J. Papastavrou. The dynamic and stochastic knapsack problem with random sized items. *Operations Research*, 46(1):17–35, 2001.

[15] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. Technical Report 97/10, DEIS, University of Bologna, DIKU, University of Copenhagen, Copenhagen, Denmark, 1997.

[16] S. Martello and P. Toth. *Knapsack Problems Algorithms and Computer Implementations*. Wiley-Interscience, New York, NY, 1990.

[17] D. Morales, J. Roda, F. Almeida, C. Rodriguez, and F. Garcia. Integral knapsack problems: Parallel algorithms and their implementations on distributed systems. In *Proceedings of the 9th International Conference on Supercomputing*, pages 218–226, Barcelona, Spain, 1995.

[18] J. Papastavrou, S. Rajagopalan, and A. Kleywegt. The dynamic and stochastic knapsack problem with deadlines. *Management Science*, 42:17061718.

[19] D. Pisinger. Core problems in knapsack algorithms. Technical Report 94/26, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, 1994.

[20] V. Poirriez, N. Yanev, and R. Andonov. A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization*, 6:110–124, 2009.

[21] D. Scarpazza, O. Villa, and F. Petrini. Efficient breadth-first search on the cell/be processor. *Parallel and Distributed Systems, IEEE Transactions on*, 19(10):1381 –1395, oct. 2008.

[22] H. Taha. *Operations Research An Introduction*. Pearson, Upper Saddle River, NJ, 2007.

[23] G. Tan, N. Sun, and G. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 135–144, San Diego, California, 2007.

[24] S. N. Vadlamani and S. F. Jenks. The synchronized pipelined parallelism model. In *The 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004.

[25] O. Villa, D. Scarpazza, F. Petrini, and J. Peinador. Challenges in mapping graph exploration algorithms on advanced multi-core processors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –10, 26-30 2007.

[26] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP algorithms. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

VITA

Hammad A. Rashid was born in Greenwich, England on April 6th, 1982, the son of Rashidullah and Nasra Rana. After completing his high school at St Patricks High School in Karachi, Pakistan, he entered Austin Community College. In the fall of 2002, he transferred to the University of Minnesota, Duluth. In the fall of 2003 he transferred to the University of Texas, Austin and received the degree of Bachelor of Science from the University of Texas, Austin in August 2006. He later worked at ClearCube, Austin, TX until August 2007. In spring 2008, he entered the Graduate College of Texas State University-San Marcos. He intends to graduate in the summer of 2010, with the degree of Master of Science.

Permanent Address: 606 W 17th st Apt 310

Austin, Texas 78701

This thesis was typed by Hammad A. Rashid.