International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878 (Online), Volume-2 Issue-1, March 2013

An Efficient Parallel Algorithm for Solving the 3-Partition Problem Based On ADI

Tahere Panahi, Tahere Heidari, Vahid Sattari Naeini

Abstract— The three-partition problem is one of the most famous strongly NP-complete combinatorial problems. Most of the recently proposed computational methods for solving partial differential equations on multiprocessor architectures stem from the 'divide and conquer' paradigm and involve some form of domain decomposition. For those methods which also require grids of points or patches of elements, it is often necessary to explicitly partition the underlying mesh, especially when working with local memory parallel processors. In this paper, a family of cost-effective algorithms for the automatic partitioning of arbitrary two- and three-dimensional finite element and finite difference meshes is presented and discussed in view of a domain decomposed solution procedure and parallel processing. We introduce properties which, in many cases, can allow either a quick solution of an instance or a reduction of its size. The average effectiveness of the properties proposed is tested through computational experiments. In this paper we propose a new approach to organize a parallel computing for finding all solutions of a problem, whose sequential algorithm takes too long finding all solutions. The parallel computing organization above presented is an combination of the bottom-up design and the divide and conquer design. We also propose a new efficient and simple algorithm for the 3-partition problem and paralellize the algorithm.

Keywords— three-Partition problem, Dynamic programming, NP complete, Divide.

I. INTRODUCTION

In the *Three-Partition Problem* (3-PARTITION) we are given a positive integer b and a set $N = \{1, 2, \ldots, n\}$ of $n = 3m$ elements, each having a positive integer *size* aj . The problem is to determine whether a partition of N into m subsets, each containing exactly three elements from N and such that the sum of the sizes in each subset is b, exists. The solution is *yes* if such a partition exists, and *no* otherwise.

This is one of the most famous NP-complete problems and, to our knowledge, the first one which was shown to be strongly NP-complete with a non trivial proof [2]. Indeed an instance I of 3-PARTITION can be solved in polynomial time if the magnitude of the largest integer value, $\mu(I)$, is bounded by a constant, but it remains NP-complete if $\mu(I)$ is bounded by a nonconstant polynomial in the instance size[3].

Revised Manuscript Received on 30 March 2013.
* Correspondence Author
Tahere Panahi, Department of Computer Engineering, Kerman Branch, Islamic Azad University, Kerman, Iran.
Tahere Heidari, Department of Computer Engineering, Kerman Branch, Islamic Azad University, Kerman, Iran.
Vahid Sattari Naeini, Department of Computer Engineering, University of Kerman ,Afzalipoor Square, Kerman, Iran.

© The Authors. Published by Blue Eyes Intelligence Engineering and Sciences Publication (BEIESP). This is an open access article under the CC-BY-NC-ND license http://creativecommons.org/licenses/by-nc-nd/4.0/

3-PARTITION has been frequently used in the literature to prove strong NP-completeness results, especially, right from the beginning, in the area of scheduling theory [4]. In this paper we study combinatorial properties of 3-PARTITION .

II. PROBLEM TRANSFORMATION

An alternative formulation of 3-PARTITION is frequently encountered in the literature: it is not explicitly required that each subset contain three elements, but the sizes are restricted to values satisfying

$b/4 < a_j < b/2 \quad (j=1,2,\ldots,n)$

An instance with unrestricted sizes can be transformed into an equivalent instance satisfying by defining

$a'_j = a_j + b \quad (j=1,\ldots,n) \quad \text{and} \quad b'=4b$

It is immediately seen that there is a one-to-one correspondence between feasible partitions of the two instances. This provides a way of transforming a general instance of 3-PARTITION into an equivalent instance of classical combinatorial optimization problems, as shown below.

Given n elements, each having a positive size, and an unlimited number of bins of identical capacity, the Bin Packing Problem (BPP) is the optimization problem consisting in the assignment of all the elements to the minimum number of bins in such a way that the total size in each bin does not exceed its capacity. Given any instance of 3-PARTITION, defined by (a_j) and b, we can define an instance of BPP having the sizes a'_j and the bin capacity b' given by (2): if the optimal solution to BPP requires exactly m bins, then we know that the answer to the 3-PARTITION instance is yes, and otherwise it is no. (Observe that this would not be true for a BPP instance defined by unrestricted sizes a_j and bin capacity b, since a solution requiring m bins could pack more or less than three elements in some bin.) Since effective exact and approximation algorithms and codes are available for BPP. this transformation gives a possible tool for the solution of 3-PARTITION.

An alternative transformation (“dual” to the previous one) can be obtained through multiprocessor scheduling. Given n jobs, each having a positive processing time, and m parallel identical processors which can perform one job at a time, the Multiprocessor Scheduling Problem (usually denoted by P||Cmax in the scheduling literature, see, e.g., Hoogeveen et al., 1997) requires an assignment of the jobs to the processors, which minimizes the largest completion time of a job (makespan). An instance of 3-PARTITION has answer yes if and only if the optimal solution to the P||Cmax instance defined by processing times a'_j ($j = 1,\ldots, n$) has makespan equal to b'. An effective exact algorithm for P||Cmax was presented by Dell’Amico and Martello (1995).

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)
© Copyright: All rights reserved

A 3-partition problem instance consists of 3q numbers, that should be divided into q groups of 3 all having the same sum. Take for example the following problem instances:

- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100, 11
- 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 200, 22
- 0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 20

For disambiguation, we write a 0 symbol do distinguish between equal numbers. In these examples, only the first instance has a solution, namely {1, 7, 11}, {2, 8, 9}, {3, 6, 10}, {4, 5, 100}. Note that the sum of each group is 19, and every number is sed exactly once. All 12 numbers in the second instance sum to 156.

This means that they should be split into four groups with a sum of 39 each (since 4 * 39 = 156). Since there are no odd numbers, this is not possible. In the third case, the sum per group should be 30 (again: 4 * 30 = 120, which is the total sum). The number 5 in it cannot be combined with two others in such a way that it forms a group of sum 30 (the reader may verify this). We introduce some definitions in order to make observations such as these in a more systematic way.

The 3-partition problem is an example of a decision problem. A decision problem is described by an instance, and a notion of feasibility. A decision problem can be seen as a yes or no question, where “yes” means that the problem is feasible, and “no” means it is not. In this paper, we mean by feasible that the problem has a solution.

Figure 1 show Young diagrams associated to the partitions of the positive integers 1 through 8. They are arranged so that images under the reflection about the main diagonal of the square are conjugate partitions.

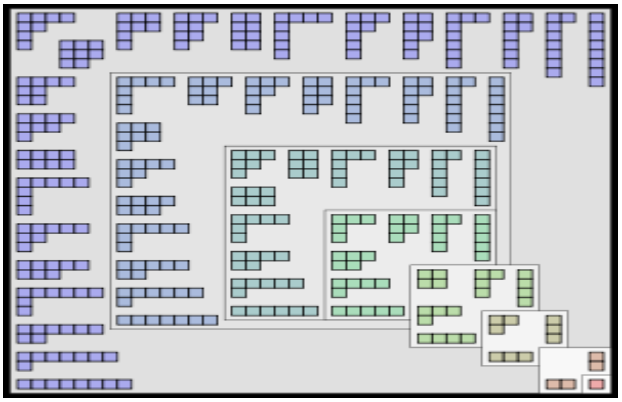

Fig1. Young diagrams associated to the partitions of the positive integers 1 through 8

III. FULL ACCOUNT

We have seen the partition problem before. The input is a collection, C, of integers, and we are interested in a subset whose sum is exactly half of the total sum of C. The problem is NPhard, and we know that splitandmerge can solve instances with about 40 numbers. But what if we have n = 500 numbers? Obviously, splitandmerge does not work anymore. We need some extra information if want to have any hope of solving this in reasonable time. Suppose that that we do have some information. we know that the sum of all the numbers is at most N = 10000. This little detail makes the problem solvable in O(nN) time.

In the spirit of dynamic programming, we will create a boolean array T of size N+1. After the algorithm has finished, T[x] will be true if and only if there is a subset of the numbers that has sum x.

Once we have that, we can simply return T[N/2]. If it is true, then there is a subset that adds up to half the total sum. To build this array, we set every entry to false to start with. Then we set T[0] to true – we can always build 0 by taking an empty set. If we have no numbers in C, then we are done! Otherwise, we pick the first number, C[0]. We can either throw it away or take it into our subset. This means that the new T[] should have T[0] and T[C[0]] set to true. We continue by taking the next element of C.

In general, suppose that we have already taken care of the first i elements of C. Now we take C[i] and look at our table T[]. It has a 1 in every location that corresponds to a sum that we can make from the numbers we have already processed. Now we add the new number, C[i]. What should the table look like? First of all, we can simply ignore C[i]. This means that no ones should disappear from T[] – we can still make all those sums. Now consider some location of T[j] that has a 1 in it. It orresponds to some subset of the previous numbers that adds up to j. If we add C[i] to that subset, we will get a new subset with total sum j+C[i]. So we should set T[j+C[i]] to true as well. That's all.

Partition: simple version

```
bool T[10240];
bool partition( vector< int > C ) {
// compute the total sum
int n = C.size();
int N = 0;
for( int i = 0; i < n; i++ ) N += C[i];
// initialize the table
T[0] = true;
for( int i = 1; i <= N; i++ ) T[i] = false;
// process the numbers one by one
for( int i = 0; i < n; i++ )
for( int j = N C[
i]; j >= 0; j)
if( T[j] ) T[j + C[i]] = true;
return T[N / 2];
}
```

Note one important detail in the j loop. It goes through the table from right to left. We have to do this in order to avoid doublecounting C[i]. For each true entry in the table, we set another entry to the right of it to true. We wouldn't want to stumble across that new entry and process it again in the same loop – that would correspond to adding C[i] to the subset twice.

We can optimize this code quite a bit. First of all, it doesn't really make sense to scan the whole table every time. All we care about is finding all the true entries. At the beginning, only the 0th entry is true. If we keep the location of the rightmost true entry in a variable, we can always start at that spot and go left instead of starting at the right end of the table. To take full advantage of this, we can also sort C[] first. That way, the rightmost true entry will move to the right as slowly as possible. Finally, we don't really care about what happens in the right half of the table (after T[N/2]) because if T[x] is true, then T[N-x] must also be true eventually – it corresponds to the complement of the subset that gave us x. Here is the optimized version.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)
© Copyright: All rights reserved

Partition: optimized version

```
bool T[10240];
bool partition( vector< int > C ) {
// compute the total sum and sort C
int n = C.size();
int N = 0;
for( int i = 0; i < n; i++ ) N += C[i];
sort( C.begin(), C.end() );
// initialize the table
T[0] = true;
for( int i = 1; i <= N; i++ ) T[i] = false;
int R = 0; // rightmost true entry
// process the numbers one by one
for( int i = 0; i < n; i++ ) {
for( int j = R; j >= 0; j)
if( T[j] ) T[j + C[i]] = true;
R = min( N / 2, R + C[i] );
}
return T[N / 2];
}
```

After the optimizations, the running time is still O(nN), but we have avoided doing a lot of useless work.

Figure 2 shows an example of partitioned indexes.

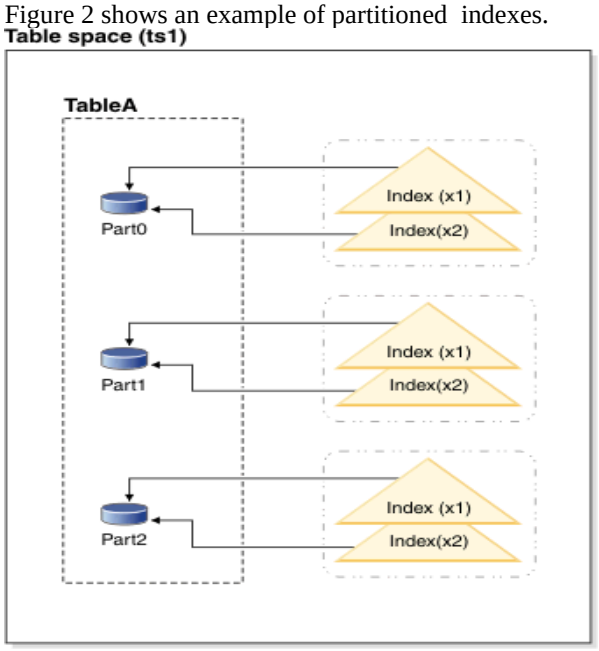

Fig2. an example of partitioned indexes.

In this example, all of the data partitions for table A and all of the index partitions for table A are in a single table space. The index partitions reference only the rows in the data partition with which they are associated. (Contrast a partitioned index with a no partitioned index, where the index references *all* rows across *all* data partitions). Also, index partitions for a given data partition are in the same index object. This particular arrangement of indexes and index partitions would have been established with statements like the following:

```
CREATE TABLE A (columns) in ts1
    PARTITION BY RANGE (column expression)
    (PARTITION PART0 STARTING FROM constant
ENDING constant,
     PARTITION PART1 STARTING FROM constant
ENDING constant,
     PARTITION PART2 STARTING FROM constant
ENDING constant,
```

```
CREATE INDEX x1 ON A (...) PARTITIONED;
CREATE INDEX x2 ON A (...) PARTITIONED;
```

IV. THE SULOTION FOR 3-PARTITION PROBLEM

For solving 3PART directly, one can either use heuristics [2], or use methods that are slower than O(nk) for any k. Calculating the candidate sets C as in equation (2.1), we have transformed the problem into an exact set-cover problem. There are efficient implementations for solving set-cover problems, for example the "Dancing Links" algorithm [12]. In this section, we look at how a SAT-solver would solve the problem, and at how an ILP-solver would. Since implementations of such solvers vary, we don't give complete descriptions or algorithms, but show an example run based on the 3PART problem given by the weights:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100, 11

To get to (2.2), we define and number the candidate sets:

Table1. an example run based on the 3-PART problem

number	{b	b'	b"}
1	1	7	11
2	1	8	10
3	1	8	10'
4	2	7	10
5	2	7	10 '
6	2	8	9
7	3	5	11
8	3	6	10
9	3	6	10'
10	3	7	9
11	4	5	10
12	4	5	10'
13	4	6	9
14	4	7	8
15	5	6	8

According to this numbering, we are looking for a vector ~y 2 $\{0, 1\}_{15}$. An ILP solver would try to find a vector y for which (2.2) holds having ~y 2 $[0, 1]_{15}$. This can be done in polynomial time, but the resulting ~y might not be integer. Take for example:

$y_{1'} = y_{3'} = y_{5'} = y_{6'} = y_{7'} = y_{8'} = y_{11'} = y_{13'} = 1/2$

$y_{i'} = 0$ all other i

To find an integer solution, the ILP-solver can try two things:

1. Try adding $y_1 = 1$ as a constraint and solve again. If there is a solution, and it is integer, we are done. If there is no solution, we have deduced y1 = 0. If there is a solution, but it is not integer, do the same trick on the next non-integer variable. This method is called branching. ILP solvers have heuristics for selecting non-integer variables.

2. Add a constraint such as:

Y'8 + y'11 + y'13 <= 1

We know this constraint must hold since the three sets mutually share an element. However, the constraint does not hold in the solution found, so we add it and try to solve our problem again. If no more such constraints can be added, and the problem is still feasible, then there are integer solutions.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)
© Copyright: All rights reserved

This step is called adding a cutting plane, since the inequality Y'8+ y'1+ y'13<= 1 can be drawn as a hyperplane in the solution-space of y'.

If we wish to solve our problem by a SAT-solver, we should use Boolean variables instead of integer variables. We choose to formulate this as:

$\vee$ p_C for all e ϵ A,
C3e

$\neg p_C \vee \neg p_{C'}$ for all C,C' ϵ C for which $C \cap C' \neq \phi$;

Note that this formulation is at worst quadratic in |C|. A SAT-solver takes a logical formula as its input. By taking the conjunction of (2.3), we obtain such a logical formula in conjunctive normal form.

The formula has a solution if and only if the original 3PART instance is feasible.

Note that the formula is of a very specific normal form: it is the conjunction of clauses with either only positive terms, or exactly two negative terms. Since 3PART is NP-complete, a SAT problem of this specific form is NP-complete as well. The proposed technique If this problem is to be solvable; then sum(ALL)/3 must be an integer. Any solution must have SUM(J) + SUM(K) = SUM(I) + sum(ALL)/3. This represents a solution to the 2-partition problem over concat(ALL, {sum(ALL)/3}). You say you have a 2-partition implementation: use it to solve that problem. Then (at least) one of the two partitions will contain the number sum(ALL)/3 - remove the number from that partion. For the other partition, run 2-partition again, to split J from K; after all, J and K must be equal in sum themselves.

Edit: This solution is probably incorrect - the 2-partition of the concatenated set will have several solutions (at least one for each of I, J, K) - however, if there are other solutions, then the "other side" may not consist of the union of two of I, J, K, and may not be splittable at all.

Try 2: Iterate over the multiset, maintaining the following map: R(i,j,k) :: Boolean which represents the fact whether up to the current iteration the numbers permit division into three multisets that have sums i, j, k. I.e., for any R(i,j,k) and next number n in the next state R' it holds that R'(i+n,j,k) and R'(i,j+n,k) and R'(i,j,k+n). Note that the complexity (as per the excersize) depends on the magnitude of the input numbers; this is a pseudo-polynomialtime algorithm. Nikita's solution is conceptually similar but more efficient than this solution since it doesn't track the third set's sum: that's unnecessary since you can trivially compute it.

Data Structures. Let T(p) stand for {q ∈ Q | (p, q) ∈ T}, i.e., T(p) is the set of states to which there is a transition from p. We also define T−1(p) to be {q ∈ Q | (q, p) ∈ T}.

The current partition is represented as an array PARTITION. It is an array of size n with (block id, state) pairs. For example, a pair (i, q) represents that the state q currently belongs to the ith block. We maintain the array PARTITION such that states belonging to the same block appear consecutively.

π := π0; split := true
while split do
split := false; let π = {B1, B2, . . ., B_}
Unmark B1, B2, . . ., B_
1. for i := 1 to n in parallel do
TEMP[i] := TSIZE[PARTITION[i].state]
2. Compute the prefix sums of TEMP[1], TEMP[2], . . . , TEMP[n]
Let the sums be v1, v2, . . . , vn
3. for i := 1 to n in parallel do
si := PARTITION[i].state
Let T[si] be {q1, . . . , qk}
for j := 1 to k in parallel do
Let processor in-charge of transition (si, qj) write
(B[si], V [si], B[qj]) in L[vi−1 + j]
4. Sort the sequence L in lexicographic order.
5. for i := 1 to m in parallel do if L[i] = L[i + 1]then L[i] := 0
6. Compress the list L using a prefix computation
7. for each block Bi (1 ≤ i ≤ %) in parallel do
for each j, 2 ≤ j ≤ ni in parallel do
if [qi,j] _= [qi,1] then mark Bi
8. if there is at least one marked block then
split := true; % := % + 1
Pick one of the marked blocks (say Bi) arbitrarily
for each p in Bi do
if [p] _= [qi,1] then
B[p] := % + 1
Change the corresponding entry in PARTITION to (p, % + 1) /* B_+1 := Bi − {p ∈ Bi : [p] = [qi,1]} and Bi := Bi − B_+1 */
Using a prefix computation, modify PARTITION such that all tuples corresponding to the same block are in successive positions.

When the array PARTITION is modified, positions of some states q's might change; inform the processors associated with the corresponding T(q)'s of this change.

Reduction 3-partition problem
with a few hints about how the proof of correctness goes.

Let the input to the reduction be x1,…,x3n,B∈Z, an instance of 3-PARTITION. Verify that $\sum i\in[3n]xi=nB$. Let β be a large number to be chosen later. For every i∈[3n] and every j∈[n], output two numbers
xiβj+βn+j+β2n+i+β(i+4)n+jβ(i+4)n+j.

Intuitively, the first number means that xi is assigned to 3-partition j, and the second number means the opposite. The xiβj term is used to track the sum of 3-partition j. The βn+j term is used to track the cardinality of 3-partition j. The β2n+i term is used to ensure that each xi is assigned exactly once. The β(i+4)n+j term is used to force these numbers into different balanced partitions.

Output two more numbers
$1+\sum j\in[n]((n-2)B\beta j+(3n-6)\beta n+j)+\sum i\in[3n](n-2)\beta 2n+i1$.

The first number identifies its balanced partition as "true", and the other, as "false". The 1 term is used to force these numbers into different balanced partitions. The other terms make up the difference between the sum of a 3-partition and the sum of its complement and the size of a 3-partition and the size of its complement and the number of times xi is assigned.

V. CONCLUSION

In this paper, An efficient parallel algorithm is proposed for solving the 3-partition problem based on ADI. The proposed technique helps to Combining these optimization algorithms and improving techniques. In this paper we propose a new approach to organize a parallel computing for finding all solutions of a problem, whose sequential algorithm takes too long finding all solutions.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)
© Copyright: All rights reserved

The parallel computing organization above presented is an combination of the bottom-up design and the divide and conquer design. We also propose a new efficient and simple algorithm for the 3-partition problem and paralellize the algorithm. The new models are practical. We also showed that the algorithm works very well for a class of 3-partition problem. The method is further enhanced through the use of a partitioning technique. In future, we will apply this technique to other problems, namely time-series processing, time-series matching, scheduling problem and system control …

REFERENCES

1. Dell'Amico, M. and Martello, S. (1995). Optimal scheduling of tasks on identical parallel processors, ORSA Journal on Computing, vol. 7, pp. 191–200.
2. Garey, M.R. and Johnson, D.S.(1975). Complexity results for multiprocessors scheduling under resource constraints, SIAM Journal on Computing, vol. 4, pp. 397–411.
3. Kern ,W. and Qiu, X. (2011). Improved taxation rate for bin packing games. In A. Marchetti-Spaccamela and M. Segal, editors, First International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems, TAPAS 2011, Rome, Italy, volume 6595 of Lecture Notes in Computer Science, PP. 175–180.
4. Ghaddar, B. Anjos, M. and Liers, F. (2008). A branch-and-cut algorithm based on semidefinite programming for the minimum k-partition problem. Annals of Operations Research, PP. 1–20.
5. Brendan,D. McKay. nauty User's Guide. (2009). Australian National University.
6. N. J. A. Sloane. Sloane's series.(2007). Number of isomorphism classes of connected 3-regular loopless multigraphs of order 2n.
7. Donald E. Knuth. Dancing links, November(2000). Technical report, Stanford University.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)
© Copyright: All rights reserved