2016 Second International Conference on Computational Intelligence & Communication Technology

Process Assignment in Multi-core Clusters Using Job Assignment Algorithm

Chapram Sudhakar, Pankaj Adhikari, T Ramesh
Department of Computer Science & Engineering
National Institute of Technology
Warangal, INDIA
e-mail: chapram@nitw.ac.in, pankajadhikari01@gmail.com, rmesht@nitw.ac.in

***Abstract*—Modern high performance cluster systems for parallel processing are employing multi-core processors and high speed interconnection networks. Efficient mapping of the processes of a parallel application onto cores of such a cluster system, plays a vital role in improving the performance of that application. Parallel application can be modelled as a weighted graph showing the communication among the processes of that application. Such a graph can be constructed with the help of profiling tools. Cluster hardware also can be modelled as a graph, by collecting hardware details using HWLOC tool. Maximum weight matching based approach can be used to embed the application graph into cluster hardware graph. The proposed approach is implemented under a cluster system and tested using benchmark MPI parallel application. The performance of the parallel application, which is mapped using the proposed approach is better than, that is mapped using the legacy packed and round robin approaches of MPI library.**

Keywords—MPI; Process Assignment; Multi-core Systems; Parallel Applications

I. INTRODUCTION

Even though the speed of high performance clusters is increasing, efficiency and running time of the parallel applications need to be improved always. As multi-core clusters are generally used to run these parallel applications, a suitable and smart placement of processes based on their communication behaviour can significantly improve the application performance. Furthermore, the underlying cluster hardware properties such as bandwidth between the processors, cache locality etc., must be considered, for improving the performance of message passing calls.

Various methods were proposed in the literature to improve the performance of parallel applications on multi-core clusters. One approach is to optimize the MPI communication performance. This approach is followed in MagPIe[1] and MPI/SX[2]. In MagPIe the collective communication operations are optimized where as in MPI/SX point-to-point communication operations are optimized. Another approach is to map parallel processes to suitable processors as in [3,4], so that communication cost can be reduced. In order to solve the mapping problem, graph partitioning techniques are used by some systems such as Chaco[5], METIS[6] and SCOTCH[7], and Hungarian method is used by Harold Kuhn [8, 9].

This paper targets process placement to tackle the locality problem that comes from the way data is exchanged between processes of a parallel application either through the network or through the memory. Job assignment method is used to find a suitable mapping for the MPI processes based on their communication pattern and hardware information to decrease the inter-node communication and thus increasing the efficiency and throughput. The processes with large inter-process communication cost are generally mapped to the same node to share the same locality or to the nodes that are close to each other so that the communication cost is reduced.

Rest of the paper is organized as follows. Section 2 gives related work done on the job assignment problem. In section 3, the proposed method is described. Section 4 discusses about the design and implementation of the algorithm. Analysis of the experimental results is presented in Section 5. The paper is summarised and concluded in section 6.

II. BACKGROUND

In MagPIe [1] and MPI/SX [2], the authors optimize the collective communications and point-to-point communications respectively. In [1], the author developed a modified library of MPI collective communication primitives (*barrier*, *broadcast*, *reduce* and *gather*), that are optimized for wide area systems by minimizing the amount of data communicated over the links. The work in [2] optimizes the point-to-point communication operations, considering the hardware topology information, by virtually ranking the MPI processes, so that more communicating pairs of processes are kept physically close to each other.

In MPIPP [3] and TREEMATCH [4] the authors try to map the parallel processes to the suitable processors in order to minimize the total communication cost across the processors of the cluster by using the locality concept and placing the extensively communicating processes in the same locality. Both works are based on the profile guided approach to obtain a suitable arrangement of the processes. MPIPP [3] finds the machine on which a process can be placed not the processor, thus not fully optimizing the placement process. TREEMATCH [4] uses algorithms for finding the independent set of minimum weights from the group of processes. As the procedure is NP-Hard, few heuristics are used for optimization.

In general, the mapping problem can be modelled as graph embedding problem which can easily be solved by graph partitioning. These techniques are used by various authors in Chaco [5], METIS [6] and SCOTCH [7].

The job assignment problem is one of the fundamental optimization problems in mathematics. It consists of finding a minimum weight perfect matching (or maximum weight matching) in a weighted bipartite graph. The Hungarian Algorithm is one of the primal simplex algorithms given by Harold Kuhn in [8, 9] that have been used to solve the linear assignment problem within time bounded by a polynomial expression of the number of processes.

III. JOB ASSIGNMENT ALGORITHM BASED METHOD

The overall placement of a process is divided into three steps – *(A)* profiling of parallel application, *(B)* gathering hardware information and *(C)* apply job assignment algorithm to obtain suitable mapping.

A. *Profiling of Parallel Application*

Using a simple profiling program, a very fine grained event trace of a parallel application can be collected, which can be used for performance analysis and optimization of that application. Many profiling programs have already been integrated into *Open MPI*. Using such profilers, information about a parallel application can be gathered, which includes number and size of messages passed between two processes, utilization of capacity of communication links between processors and amount of computation done by the processes etc. A common cost (or profit) function can be defined, based on the parameters collected by the profilers, that needs to be optimized.

B. *Gathering Hardware Information*

HWLOC is a well known software tool used to gather the hardware information of a machine. This tool provides a portable abstraction of the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and processing units. It aims at helping applications by providing necessary API calls to gather information about underlying computing hardware.

C. *Mapping of the Processes*

MPI based parallel processes can be mapped in two ways - resource binding and rank reordering methods. In resource binding method, user has to select a processor to execute a process under consideration to decrease overall communication cost. Thus, each process is bound to a specific resource (processor). In rank reordering method, ranks of the processes are re-assigned based on the information obtained through step *A* and *B*. This new arrangement is used to create a new logical topology and the corresponding communicator. The new communicator is used by all of the participating MPI processes for their communication.

Given *n* tasks and *m* resources, tasks should be assigned to resources on many-to-one basis, with a predefined cost (or profit) of assigning a given task to a given resource. Finding an optimal assignment (maximize profit or minimize the cost) is known as Job Assignment Problem. In the present scenario, *n* MPI processes (tasks) are considered to assign to *m* processors (resources). The root process gathers, the communication cost for the MPI processes and underlying hardware architecture. Using the gathered information, job assignment algorithm is applied to compute the new ranks of the processes and then broadcast the new ranks among all of the MPI processes.

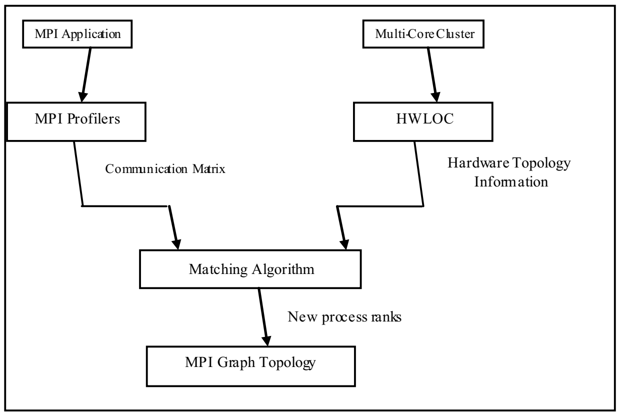

Fig. 1. Control Flow of the Algorithm

Broadly the steps of the proposed method are shown in Fig. 1. The MPI application information is obtained using MPI Profilers and the underlying hardware topology information is obtained using HWLOC. Using that information, graph matching algorithm finds the new ranks of the MPI processes, and creates a new communicator with modified MPI Graph topology.

IV. ALGORITHM AND IMPLEMENTATION

In this section input and output details and main algorithms are presented.

A. *Input*

1) *Cost Matrix*

Cost matrix is a square matrix of size *n* x *n*, where *n* is the total number of processes in the parallel application. Value of each entry corresponds to the cost of communication between two processes. The cost of communication can be based on number of messages passed between them, computational weight of each process, computational power of cores and bandwidth of the communication link etc. Table I shows an example cost matrix for 8 processes.

2) *Hardware Information File*

Hardware information includes underlying hardware architecture in hierarchical form. In modern architectures, the hardware topology can be represented as a hierarchical topology including NUMA memory nodes, sockets, shared caches, cores and processing units. HWLOC tool is used to gather the underlying hardware information and store it in the required format. In the hardware information file, first line contains *n*, which indicates number of levels in hardware architecture, and second line contains *n* integer values indicating the arity (number of children) of each level starting from root. Sample content of hardware information file is shown below, for a tree type architecture.

```
3
2    4      2
```

In this example the hardware hierarchy contains three levels. Level 1 has arity 2, which means that the root of the tree has two children. Similarly, level 2 has arity 4, thus indicating 4 children for each node at level 2, and level 3 has arity 2 indicating 2 children for each node in level 3. Fig. 2 shows the pictorial representation of above example.

Generally, the leaves of the tree are processing units or cores. The MPI processes will be placed on these units. Further levels upwards consist of cache hierarchy (e.g. L1, L2 and L3), memory nodes, sockets, NUMA nodes etc.

TABLE I. SAMPLE COST MATRIX

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	100	19	12	19	11	2511	223
1	100	0	12	12	25	28	223	1412
2	19	12	0	223	251	1584	17	10
3	12	12	223	0	1995	112	17	17
4	19	25	251	1995	0	158	17	12
5	11	28	1584	112	158	0	25	19
6	2511	223	17	17	17	25	0	100
7	223	1412	10	17	12	19	100	0

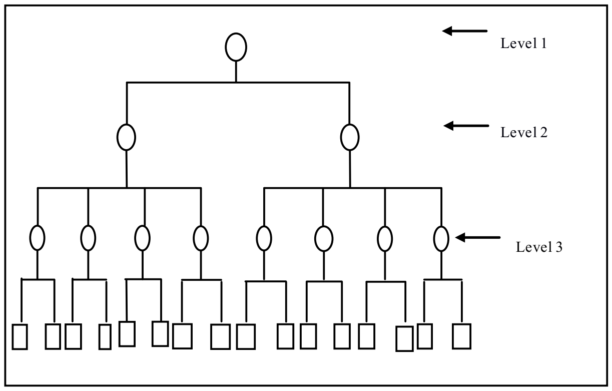

Fig. 2. Sample Hardware Architecture

B. Algorithm

The goal of the proposed algorithm is to minimize the overall cost for the application to run in parallel. So the processes with more inter process communication cost may be assigned to the same core. If not possible then such processes are assigned to the nearest processing units. The Hungarian method [10] depicted in Fig. 3 is used to evaluate the matching of the processes.

In [11] James Munkres proved that the Hungarian algorithm is strictly polynomial. The original implementation of the algorithm has the complexity of $O(n^4)$, where n is the number of processes. However, it has been proved that the original algorithm can be modified to $O(n^3)$ by Edmonds and Karp.

Fig. 4 shows the algorithm used to calculate the new arrangement for the processes. In this, the processes are iteratively combined based on their communication cost using graph matching techniques. In order to minimize the total communication cost, the processes having higher inter-process communication cost are paired first. After finding an optimal matching, the aggregated new communication matrix is computed. This new communication matrix is used in the next iteration.

1. Subtract the smallest entry in each row from all the entries of its row.
2. Subtract the smallest entry in each column from all the entries of its column.
3. Draw lines through appropriate rows and columns so that all the zero entries of the cost matrix are covered and the minimum number of such lines is used.
4. Test for Optimality:
   a. If the minimum number of covering lines is equal to the number of processes, an optimal assignment of zero's is possible, then the algorithm is completed.
   b. If the minimum number of covering lines is less than the number of processes, an optimal assignment of zeros is not yet possible. In that case, proceed to Step 5.
5. Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Return to Step 3.

Fig. 3. Hungarean Assignment Algorithm

```
Mapping Algorithm
Input : CM - Cost matrix
N – Size of cost matrix
arity – array storing arities of different levels
Output : new_result – assignment result

old_result = NULL
for i -> num_of_level to 0 down by 1
 for j -> 1 to arity_at_level_i up by j*2
   result = assignment(CM, N, arity)
   aggregated_mat = aggregatematrix(CM,
                              result, N)
   if old_result is not NULL
     new_result = aggregateresult(old_result,
                           new_result, N)
   oldresult = new_result
   CM = aggregated_mat
 end loop j
end loop i
```

Fig. 4. Mapping Algorithm

For the example cost matrix given in Table I and with hardware interconnection network of 3-level binary tree structure, the result of the first iteration of mapping algorithm is (0, 6), (1, 7), (2, 5) and (3, 4). Thus, after calling Mapping Algorithm, the resultant aggregated new cost matrix is shown in Table II. Algorithm used for computing aggregate cost matrix is shown Fig. 5.

TABLE II. COST MATRIX AFTER ITERATION 1

	0	1	2	3
0	0	646	72	65
1	646	0	69	66
2	72	69	0	744
3	65	66	744	0

In this, the process zero is the group of processes (0, 6), process one is the group of processes (1, 7) and so on. The value corresponding to (0, 1) in new cost matrix is evaluated by combining the total cost between (0, 1), (0, 7), (6, 1) and (6, 7), i.e., 100, 223, 223 and 100 respectively in the original cost matrix. Thus, the new cost must be the collective cost between these two groups. Similarly, for the next iteration the result of mapping algorithm is (0, 1) and (2, 3)

```
Aggregate Cost Matrix
Input: old_matrix, old_result, n, arity
Output: new aggregated_matrix
  allocate and initialize new_matrix
  for i->0 to n/arity increment by 1
    for j->0 to n/arity increment by 1
      new_matrix[i][j] = 0
      if i is not equal to j
        for k->0 to arity increment by 1
          for p->0 to arity increment by 1
            l = old_result[i][k]
            m = old_result[j][p]
            new_matrix[i][j] += old_matrix[l][m]
          end for
        end for
    end for
  end for
```

Fig. 5. Aggregate Cost Matrix

The new process groups from third iteration are (0, 6, 1, 7) and (2, 5, 3, 4). The corresponding cost matrix is shown in Table III. Thus, the new arrangement for placing the processes is (0, 6, 1, 7, 2, 5, 3, 4) in place of (0, 1, 2, 3, 4, 5, 6, 7). The output of Mapping Algorithm is then used as an input to the algorithm for re-assignment of process ranks which is shown in Fig. 6.

TABLE III. COST MATRIX AFTER ITERATION 2

	0	1
0	0	272
1	272	0

Using this modified arrangement information, while re-assigning the process ranks, a new communicator with new communication topology is created. Thus process ranks are reordered using MPI_Dist_graph_create(...) library call. This call is made just after the initialization step and before any application data is loaded into the MPI processes. Subsequent communication operations are performed using this new communicator which gives better performance.

```
Re-Assignment of Process Ranks
Input: file - adjacency_matrix
         file – cost_matrix
Output: new communicator for MPI processes
  Initialize MPI
  if my_rank is 0
    initialize array pointers: sources, degrees,
        weights and destinations as NULL
    enter_graph_values ( num_prcs,
        destinations, sources, degrees,
        adjacency_matrix)
    MPI_Dist_graph_create(MPI_COMM_WORLD,
        comm_size, sources, degrees,
        destinations, MPI_UNWEIGHTED,
        MPI_INFO_NULL, reorder,
        &comm_topology)
  else
     MPI_Dist_graph_create(MPI_COMM_WORLD,
        0, NULL, NULL, NULL, MPI_UNWEIGHTED,
        MPI_INFO_NULL, reorder,
        &comm_topology)

  if comm_topology != MPI_COMM_NULL
     comm_to_use = comm_topology
  else
     comm_to_use = MPI_COMM_WORLD

  MPI_Comm_rank ( comm_to_use, &my_rank)
  Rest of the MPI Code
```

Fig. 6. Re-Assignment of Process Ranks

The MPI process manager generally schedules the processes in a round robin or packed manner. But because of rank reordering, the process manager has to use the new ranks which have been assigned under the new communicator.

C. Output

The output of Mapping Algorithm is stored in a file in the following format. The file contains n+ 1 lines.

n
A_1 a_{11} a_{12} a_{13}....
A_2 b_{11} b_{12} b_{13}....
....
....
A_n c_{11} c_{12} c_{13}....

The first line contains the number of nodes (processes) of the graph. The next *n* lines represent the information for *n* processes. The *i*-th process will have information like

$A_i \quad d_{11} \quad d_{12} \quad d_{13} \ldots$

In this, first integer represents the degree of *i*-th node. The rest of the line contains A_i integers, each integer indicating the node id of the neighbouring node. For the input example in the previous section, output file of Mapping Algorithm is shown below.

```
8
2   6   7
2   5   7
2   4   5
1   4
2   2   3
2   1   2
1   0
2   0   1
```

The corresponding new topology graph created by MPI_Dist_graph_create for the example given in section 4.2, is shown in Fig. 7.

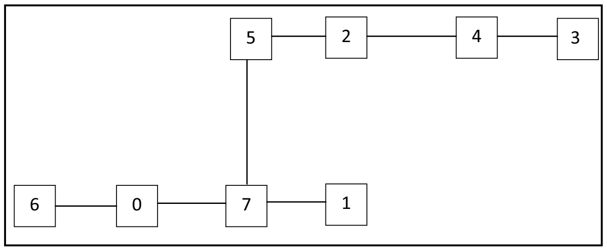

Fig. 7. Sample Topology Created by MPI_Dist_graph_create

This graph indicates the position of the old processes in new communicator. The graph clearly justifies the grouping shown in the example of section 4.2, i.e. (0, 6, 1, 7) and (2, 5, 3, 4). It also defines the edges between two groups. The edge between two groups is determined by the maximum entry in the cost matrix between the chosen processes. For example, in order to determine the edge between group (0, 6) and (1, 7), the entries numbered $C_{0,1}$, $C_{0,7}$, $C_{6,1}$ and $C_{6,7}$ are checked in the cost matrix to find the maximum value. In this example $C_{0,7}$ has the maximum value thus there is an edge between process 0 and process 7 in the adjacency graph. The other edges are identified with a similar reasoning.

V. Results

The platform used for the experimentation is a 1+8 node cluster running Rocks cluster 6.1.1 on top of CentOS 6.5. The front end of the cluster is HP Proliant DL380p Gen8, 2 x Intel Xeon E5-2640 processor having 2.5 GHz clock, 6-core, 15 MB cache and 64 GB RAM. Each node contains 1 x Intel Xeon E5-2640 Processor with 2.5 GHz clock, 6-core, 15 MB cache, 16 GB RAM and 2 * 300 GB HDD. To run the implemented algorithms the following software is used: GNU C compiler, Open MPI, HWLOC and MPI inbuilt profilers.

The application selected for comparison is IS (Integer Sort) from NASA NPB benchmark suite. This application is more communication oriented and has some regular pattern for communication. Execution time is considered as parameter of comparison for the application versions with and without reordering of the processes. The comparative results are shown in Table IV. and the corresponding line graph is shown in Fig. 8.

From table IV, it is clear that initially the proposed algorithm is taking more time for less number of processes (8 or 16) due to the overhead of calculating the suitable arrangement and topology creation. But as the number of processes increases (more than 16), the fraction of overhead time consumed by the proposed algorithm decreases gradually. The percentage of improvement in the time of execution is shown in last column of Table IV.

TABLE IV. Execution Time With and Without Rank Reordering of Processes

Number of Process	Time without reordering (seconds)	Time with Reordering (seconds)	% improvement
8	0.0225	0.05614	-149%
16	17.543	18.01022	-2.66%
32	98.201	74.32241	+24.31%
64	262.736	189.23373	+27.97%
128	617.531	425.23948	+31.13%

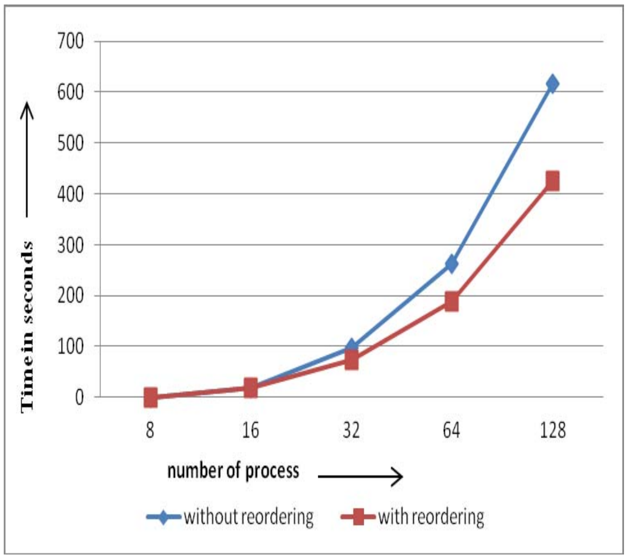

Fig. 8. Comparison of Execution Time With and Without Reordering of Processes

VI. Conclusions and Future Work

The locality problem is becoming a major challenge for current applications. Improving data access and communication is a key issue for obtaining good performance out of the underlying hardware. However, not only does the communication speed between computing units depend on their locations (due to cache size, memory hierarchy, latency and network bandwidth, topology, etc.), but also on the communication pattern between processes.

This paper, presents a new method, which computes a process placement of the application tailored for the target machine. It is based on both the application communication pattern and the architecture of the machine. Unlike other approaches using graph-partitioning techniques, this method does not need accurate and quantitative information about the various communication speeds, but it only requires qualitative information like number of messages.

Results show that the proposed method considerably decreases the execution time of the application in comparison to the MPI's legacy Packed and RR policies. It has been observed that as the number of processes increases and as the additional overhead fraction is reduced, the results are improved more.

Future work is directed towards a distributed version of this method for the use under large-scale machines and making it easier to gather communication pattern. Another study will focus on the different metrics, that can be used to further optimize, like including the node computation power and its bandwidth as a deciding factor in the cost matrix. Experiments on very large machines are also targeted, especially ones with a large number of cores.

References

[1] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, Raoul A. F. Bhoedjang, "MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems", PPoPP, Symposium on Principles and Practice of Parallel Programming, pp. 131-140, Atlanta, 1999.

[2] J. L. Traff, "Implementing the MPI process topology mechanism", Supercomputing, ACM/IEEE 2002 Conference, pp. 1–14, Baltimore, 2002.

[3] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: An Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters", Proc. 20th Ann. Int'l Conf. Supercomputing (ICS), pp. 353-360, Queensland, 2006.

[4] Emmanuel Jeannot, Guillaume Mercier, Francois Tessier, "Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques", IEEE Transaction on Parallel and Distributed Systems, vol. 25, no. 4, pp. 993-1002, 2014.

[5] B. Hendrickson and R. Leland, "The Chaco User's Guide: Version 2.0", Technical Report SAND94–2692, Sandia Nat'l Laboratory, 1994.

[6] G. Karypis, V Kumar, "METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0", technical report 1995.

[7] F. Pellegrini, "Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs", Proc. Scalable High-Performance Computing Conf. (SHPCC '94), pp. 486-493, May 1994.

[8] Harold W. Kuhn, "The Hungarian Method for the assignment problem", Naval Research Logistics Quarterly, pp. 83–97, 1955.

[9] Harold W. Kuhn, "Variants of the Hungarian method for assignment problems", Naval Research Logistics Quarterly, pp. 253–258, 1956.

[10] Derek Bruff, "The Assignment Problem and the Hungarian Method", Math 20 –Harvard Mathematics Department, 2005. http://www.math.harvard.edu/archive/20_spring_05/handouts/assignment_overheads.pdf

[11] J. Munkres, "Algorithms for the Assignment and Transportation Problems", Journal of the Society for Industrial and Applied Mathematics, pp. 32–38, 1957.