Original citation:

Permanent WRAP url:
http://wrap.warwick.ac.uk/71239

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher's statement:
“© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.”

Journal homepage: http://www.computer.org/web/tpds

A note on versions:
The version presented here may differ from the published version or, version of record, if you wish to cite this item you are advised to consult the publisher’s version. Please see the ‘permanent WRAP url’ above for details on accessing the published version and note that access may require a subscription.

For more information, please contact the WRAP Team at: publications@warwick.ac.uk

http://wrap.warwick.ac.uk
Developing Graph-based Co-scheduling Algorithms on Multicore Computers

Ligang He, Huanzhou Zhu and Stephen A. Jarvis
Department of Computer Science, University of Warwick, Coventry, CV4 7AL, United Kingdom
Email: {liganghe, zhz44, saj}@dcs.warwick.ac.uk

Abstract—It is common that multiple cores reside on the same chip and share the on-chip cache. As a result, resource sharing can cause performance degradation of co-running jobs. Job co-scheduling is a technique that can effectively alleviate this contention and many co-schedulers have been reported in related literature. Most solutions however do not aim to find the optimal co-scheduling solution. Being able to determine the optimal solution is critical for evaluating co-scheduling systems. Moreover, most co-schedulers only consider serial jobs, and there often exist both parallel and serial jobs in real-world systems. In this paper a graph-based method is developed to find the optimal co-scheduling solution for serial jobs; the method is then extended to incorporate parallel jobs, including multi-process, and multi-threaded parallel jobs. A number of optimization measures are also developed to accelerate the solving process. Moreover, a flexible approximation technique is proposed so that we can control the scheduling speed by setting the requirement for the solution quality.

I. INTRODUCTION

Multicore processors have become mainstream products in the CPU industry. In a multicore processor, multiple cores reside and share resources on the same chip. However, with such a design, running multiple applications on different cores can cause resource contention, which in turn leads to performance degradation [20]. Many researchers have shown that it is possible to isolate some resources, such as disk [33] and network [15] bandwidth for co-running jobs. However, it is very difficult to isolate the on-chip last level cache (LLC). This is known as the shared cache contention problem and has been studied extensively, including [18], [20], [36]. Existing approaches to addressing on-chip shared cache contention fall into three categories: 1) Architectural-level solutions to improve hardware and provide isolation among threads [24] [29], 2) System-level solutions to partition the cache for each application [34] [22], and 3) Software-level solutions to develop schedulers to reduce contention [12] [14]. Within these three categories, architectural-level solutions remain under development by processor vendors, and cache partitioning solutions require many changes to existing system-level software (such as the operating system) and therefore incur high implementation cost. The third approach, developing contention-aware schedulers, is regarded as a lightweight approach, and has therefore attracted much attention by researchers – it is this last category that is the focus of this research.

A number of contention-aware co-schedulers have been developed [16], [28], [37]. These studies demonstrate that contention-aware schedulers can deliver better performance than conventional schedulers. However, they do not aim to determine or deliver optimal co-scheduling performance. Such a benchmark is useful to understand, even if it is obtained offline, as this allows system designers to assess potential for improvement. In addition, knowing the gap between current and optimal performance can also help scheduler designers make tradeoffs between scheduling efficiency (i.e., the time that the algorithm takes to compute the scheduling solution) and scheduling quality (i.e., how good the obtained scheduling solution is).

Co-schedulers in the literature that propt to be optimal only consider serial jobs (each of which runs on a single core). For example, the work in [18] modelled the optimal co-scheduling problem for serial jobs as an integer programming problem. However, in modern multi-core systems, especially those in cluster and cloud platforms, both parallel and serial jobs co-exist [9], [17], [32]. In order to address this problem, this paper proposes a new method to determine the optimal co-scheduling solution for a mix of serial and parallel jobs. Two types of parallel jobs are considered in this paper: Multi-Process Parallel (MPP) jobs, such as MPI jobs, and Multi-Thread Parallel (MTP) jobs, such as OpenMP jobs. In this paper, we first propose a method to co-schedule MPP and serial jobs, and then extend this method to handle MTP jobs.

Resource contention presents different features in single processor and multi-processor machines. In this paper, a layered graph is first constructed to model the co-scheduling problem on single processor machines. The problem of finding the optimal co-scheduling solutions is then modelled as finding the shortest valid path in the graph. Further, this paper develops a set of algorithms to find the shortest valid path for both serial and parallel jobs. A number of optimization measures are also developed to increase the scheduling efficiency of these proposed algorithms (i.e., to accelerate the solving process of finding the optimal co-scheduling solution). After these, the graph model and proposed algorithms are extended to co-scheduling parallel jobs on multi-processor machines.

It has been shown that the A*-search algorithm is able to effectively avoid unnecessary searches when finding optimal solutions. In this paper, an A*-search-based algorithm is also developed to combine the ability of the A*-search algorithm and the proposed optimization measures in terms of accelerating the solving process. Finally, a flexible approximation technique is proposed so that we can control the scheduling...
efficiency by setting the requirement for the solution quality.

We conduct experiments with real jobs to evaluate the effectiveness of the proposed co-scheduling algorithms. The results show that i) the proposed algorithms can find the optimal co-scheduling solution for both serial and parallel jobs, ii) the proposed optimization measures can significantly increase the scheduling efficiency, and iii) the proposed approximation technique is effective in the sense that it is able to balance the scheduling efficiency and the solution quality.

The remainder of this paper is organized as follows. Section II discusses related work. Section III formalizes the co-scheduling problem for both serial and MPP jobs, and presents a graph-based model for the problem. Section IV presents methods and optimization measures to determine the optimal co-scheduling solution for serial jobs. Section V extends the methods proposed in Section IV to incorporate MPP jobs and presents an optimization technique for the extended algorithm. In Section VI, we extend the graph-based model and proposed algorithms for multi-processor machines; Section VII then adjusts the graph model and the algorithms for MTP jobs. Section VIII presents the A*-search-based algorithm; a clustering approximation technique is proposed in Section IX to control the scheduling efficiency according to the required solution quality. Experimental results are presented in Section X and finally, Section XI concludes the paper and presents avenues for future work.

II. RELATED WORK

This section first discusses co-scheduling strategies proposed in the literature. Similar to the work presented in [18], our method needs to know the performance degradation of the jobs when they co-run on a multi-core machine. Therefore, this section also presents methods proposed for acquiring such performance degradation information.

A. Co-scheduling strategies

Many co-scheduling schemes have been proposed to reduce shared cache contention in multi-core processors. Different metrics can be used to capture resource contention, including Cache Miss Rate (CMR), overuse of memory bandwidth, and performance degradation of co-running jobs. These schemes fall into the following two classes:

1. The first is co-scheduling schemes that aim to improve runtime scheduling by providing online scheduling solutions. The work in [6], [11], [36] develops co-schedulers that reduce the cache miss rate of co-running jobs; the fundamental idea is to uniformly distribute the jobs with high cache requirements across the processors. Wang et al. [31] demonstrate that cache contention can be reduced by rearranging the scheduling order of tasks.

2. The second class of co-scheduling schemes focuses on providing a basis for conducting performance analysis. It primarily aims to determine optimal co-scheduling performance in an offline manner, in order to provide a performance target for other co-scheduling systems. Extensive research is conducted in [18] to find co-scheduling solutions; the work models the co-scheduling problem for serial jobs as an Integer Programming (IP) problem, and then uses an existing IP solver to find the optimal co-scheduling solution. This research also provides a set of heuristic algorithms to determine near optimal co-scheduling. Co-scheduling studies in this category of solution only consider serial jobs, and primarily apply a heuristic approach to finding solutions.

The research presented in this paper falls into the second class of solution. Here a new offline method is developed to find the optimal co-scheduling solution for both serial and parallel jobs.

B. Acquiring information on performance degradation

When a job co-runs with a set of other jobs, its performance degradation can be obtained either through prediction [7], [13], [19], [35] or through offline profiling [30].

Predicting performance degradation has been well studied in the literature [7], [10], [26], [35]. One of the best-known methods is Stack Distance Competition (SDC) [7], a method which uses a Stack Distance Profile (SDP) to record the hits and misses of each cache line when each process is running alone. The SDC model tries to construct a new SDP that merges the separate SDPs of individual processes that are to be co-run together. This model relies on the intuition that a process that reuses its cache lines more frequently will occupy more cache space than other processes. Based on this, the SDC model examines the cache hit count of each process’s stack distance position. For each position, the process with
the highest cache hit count is selected and copied into the merged profile. After the algorithm reaches the last position, the effective cache space for each process is computed based on the number of stack distance counters in the merged profile.

Offline profiling can obtain more accurate degradation information, although the process itself is more time consuming. Since the goal of our research is to find the optimal co-scheduling solutions offline, this method is also applicable in our work.

III. FORMALIZING THE CO-SCHEDULING PROBLEM

First, in Subsection III-A, we briefly summarize the approach in [18] to formalizing the co-scheduling of serial jobs. Subsection III-B then formalizes the objective function for co-scheduling a mix of serial and MPP jobs, and Subsection III-C presents a graph model for the co-scheduling problem. This section focuses on single processor machines, i.e., all CPU cores reside on the same chip.

A. Formalizing the co-scheduling of serial jobs

The work in [18] shows that due to resource contention, co-running jobs generally run slower on a multi-core processor than if they run alone. This performance degradation is called the co-run degradation. When a job \( i \) co-runs with the jobs in a job set \( S \), the co-run degradation of job \( i \) can be formally defined as in Eq. 1, where \( CT_i \) is the computation time when job \( i \) runs alone, \( S \) is a set of jobs and \( CT_{i,S} \) is the computation time when job \( i \) co-runs with the set of jobs in \( S \). Typically, the value of \( d_{i,S} \) is a non-negative value.

\[
d_{i,S} = \frac{CT_{i,S} - CT_i}{CT_i}
\]

In the co-scheduling problem considered in [18], \( n \) serial jobs are allocated to multiple u-core processors so that each core is allocated one job. \( m \) denotes the number of u-core processors needed, which can be calculated as \( \frac{n}{u} \) (if \( n \) cannot be divided by \( u \), we can simply add \( (u - n \mod u) \) imaginary jobs which have no performance degradation with any other jobs). The objective of the co-scheduling problem is to find the optimal way to partition \( n \) jobs into \( m \) u-cardinality sets, so that the sum of \( d_{i,S} \) in Eq.1 over all \( n \) jobs is minimized, which is expressed in Eq. 2.

\[
\min \sum_{i=1}^{n} d_{i,S}
\]

B. Formalizing the co-scheduling of serial and parallel jobs

We first model the co-scheduling of embarrassingly parallel (PE) jobs (i.e., those with no dependency and communications between parallel processes), and then extend the model to co-schedule parallel jobs with inter-process communications (denoted by the term PC). An example of a PE job is parallel Monte Carlo simulation [27]. In such an application, multiple slave processes are running simultaneously to perform the Monte Carlo simulations. After a slave process completes its portion of work, it sends the result back to the master process. After the master process receives the results from all slaves, it reduces the final result (i.e., by calculating the average). An example of a PC job is an MPI application for matrix multiplication. In both types of parallel job, the completion time of a job is determined by the slowest process in the job.

Eq.2 cannot be used to satisfy our objective of finding the optimal co-scheduling of parallel jobs. This is because Eq.2 will sum the degradation experienced by each process of a parallel job. However, as explained above, the completion time of a parallel job is determined by its slowest process. In the case of PE jobs, a larger degradation of a process indicates a longer execution time for that process. Therefore, no matter how small the degradation other processes have, the execution flow in the parallel job must wait until the process with the largest degradation completes. Thus, the completion time of a parallel job is determined by the largest degradation experienced by all its processes; this is denoted by Eq.3, where \( d_{ij,S} \) is the degradation (measured in time) of the \( j \)-th process, \( p_{ij} \) co-runs with the job set \( S \). Therefore, if the set of jobs to be co-scheduled includes both serial jobs and PE jobs, the total degradation should be calculated using Eq. 4, where \( n \) is the number of all serial jobs and parallel processes, \( P \) is the number of parallel jobs, \( S_i \) and \( S_{ij} \) are the set of co-running jobs that includes job \( p_i \) and parallel process \( p_{ij} \). \( S_i \) and \( S_{ij} \) are then the set of jobs excluding \( p_i \) and \( p_{ij} \), respectively. Now the objective is to find such a partition of \( n \) jobs/processes into \( m \) u-cardinality sets such that Eq. 4 is minimized.

\[
\max_{p_{ij}\in P_i}(d_{ij,S})
\]

\[
\sum_{i=1}^{n}(\max_{p_{ij}\in P_i}(d_{ij,S_{ij}}-(p_{ij}))) + \sum_{i=1}^{n-P} d_{i,S_{i}-(p_{i})}
\]

In the case of PC jobs, the slowest process in a parallel job is determined by both performance degradation and inter-process dependencies (i.e., communication time). Therefore, we define the communication-combined degradation, which is expressed using Eq. 5, where \( c_{ij,S} \) is the communication time taken by parallel process \( p_{ij} \) when \( p_{ij} \) co-runs with the processes in \( S \). As with \( d_{ij,S} \), \( c_{ij,S} \) also varies with the co-scheduling solutions. We can see from Eq. 5 that for all process in a parallel job, those with the largest sum of performance degradation (in terms of computation time) and communication has the greatest value of \( d_{ij,S} \), since the computation time of all processes (i.e., \( CT_{ij} \)) in a parallel job is the same when a parallel job is evenly balanced. Therefore, the greatest \( d_{ij,S} \) of all processes in a parallel job should be used as the communication-combined degradation for that parallel job.

When the set of jobs to be co-scheduled includes both serial jobs and PC jobs, we use Eq.5 to calculate \( d_{ij,S} \) for each parallel process \( p_{ij} \), and then we replace \( d_{ij,S} \) in Eq.4 with that calculated by Eq.5 to satisfy the objective of co-scheduling a mix of serial and PC jobs.

\[
d_{ij,S} = \frac{CT_{ij,S} - CT_{ij} + c_{ij,S}}{CT_{ij}}
\]

C. A graph model for co-scheduling

In this research we propose a graph-based approach to finding the optimal co-scheduling solution for both serial and parallel jobs. In this section, the graph model is first presented
and, following this, intuitive strategies to solve the graph model are then discussed.

1) The graph model: As formalized in Section III-A, the objective of solving the co-scheduling problem for serial jobs is to find a way to partition \( n \) jobs, \( j_1, j_2, \ldots, j_n \), into \( m \) \( u \)-cardinality sets, so that the total degradation of all jobs is minimized. The number of all possible \( u \)-cardinality sets is \( \binom{n}{u} \). In this paper, a graph is constructed, called the co-scheduling graph, to model the co-scheduling problem for serial jobs (we will discuss in Section V how to use this graph model to handle parallel jobs). There are \( \binom{n}{u} \) nodes in the graph and a node corresponds to a \( u \)-cardinality set. Each node represents a \( u \)-core processor with \( u \) jobs assigned to it. The ID of a node consists of a list of the IDs of the jobs in the node. In the list, the job IDs are always placed in an ascending order. The weight of a node is defined as the total performance degradation of the \( u \) jobs in the node. The nodes are organized into multiple levels in the graph. The \( i \)-th level contains all nodes in which the ID of the first job is \( i \). At each level, the nodes are placed in ascending order of their ID’s. A start node and an end node are added as the first (level 0) and the last level of the graph, respectively. The weights of the start and end nodes are both 0. The edges between the nodes are dynamically established as the algorithm of finding the optimal solution progresses. Such organization of the graph nodes will be used to help reduce the time complexity of the co-scheduling algorithms proposed in this paper. Figure 1 illustrates the case where 6 jobs are co-scheduled to dual-core processors. The figure also shows how to code the node IDs in the graph and how to organize the nodes into different levels. Note that for clarity we do not draw all edges.

![Fig. 1: The exemplar co-scheduling graph for co-scheduling 6 jobs on dual-core machines; the list of numbers in each node is the node ID; A number in a node ID is a job ID; The edges of the same color form the possible co-scheduling solutions; The number next to the node is the node weight, i.e., total degradation of the jobs in the node.](image)

In the constructed co-scheduling graph, a path from the start to the end node forms a co-scheduling solution if the path does not contain duplicated jobs; this we call a valid path. The distance of a path is defined as the sum of the weights of all nodes on the path. Finding the optimal co-scheduling solution is equivalent to finding the shortest valid path from the start to the end node. It is straightforward to know that a valid path contains at most one node from each level in the graph.

2) Intuitive strategies to solve the graph model: We first try to solve the graph model using Dijkstra’s shortest path algorithm [8]. However, we find that Dijkstra’s algorithm cannot be directly applied to find the correct solution. This can be illustrated using the example in Figure 1. In order to quickly reveal the problem, let us consider only five nodes in Figure 1, \( \langle 1,5 \rangle, \langle 1,6 \rangle, \langle 2,3 \rangle, \langle 4,5 \rangle, \langle 4,6 \rangle \). Assume the weights of these nodes are 11, 9, 9, 7 and 4, respectively. Out of all these five nodes, there are two valid paths reaching node \( \langle 2,3 \rangle: \langle 1,5 \rangle, \langle 2,3 \rangle \) and \( \langle 1,6 \rangle, \langle 2,3 \rangle \). Since the distance of \( \langle 1,6 \rangle, \langle 2,3 \rangle \), which is 18, is shorter than that of \( \langle 1,5 \rangle, \langle 2,3 \rangle \), which is 20, the path \( \langle 1,6 \rangle, \langle 2,3 \rangle \) will not be examined again according to Dijkstra’s algorithm. In order to form a valid schedule, the path \( \langle 1,6 \rangle, \langle 2,3 \rangle \) has to connect to node \( \langle 4,5 \rangle \) to form a final valid path \( \langle 1,6 \rangle, \langle 2,3 \rangle, \langle 4,5 \rangle \) with the distance of 25. However, we can see that \( \langle 1,5 \rangle, \langle 2,3 \rangle, \langle 4,6 \rangle \) is also a valid schedule and its distance is less than that of \( \langle 1,6 \rangle, \langle 2,3 \rangle, \langle 4,5 \rangle \). However, the schedule \( \langle 1,5 \rangle, \langle 2,3 \rangle, \langle 4,6 \rangle \) is dismissed by Dijkstra’s algorithm during the search for the shortest path.

The main reason for this problem is that Dijkstra’s algorithm only records the shortest subpaths reaching a certain node and dismisses other optional subpaths. This is fine for searching for the shortest path, but in our problem we have to search for the shortest valid path. Dijkstra’s algorithm searches up to a certain node in the graph, recording only the shortest subpath up to that node. As such, not all nodes among the unsearched nodes can form a valid schedule with the current shortest subpath, which may cause the shortest subpath to connect to nodes with bigger weights. As illustrated, a subpath that has been dismissed by Dijkstra’s algorithm may be able to connect to the unsearched nodes with smaller weights and therefore generate a shorter final valid path.

In order to address the above problem, an intuitive strategy is to revise Dijkstra’s algorithm so that it will not dismiss any subpath, i.e., to allow the algorithm to record every visited subpath. Then, the path with the smallest distance among all examined and complete paths is the optimal co-scheduling result. This strategy is equivalent to enumerating all possible subpaths in the graph. The time complexity of such a strategy is very high, which will be discussed when we compare it with the SVP algorithm presented in Subsection IV-A. This time complexity motivates us to design more efficient algorithms to find the shortest valid path. In the next section, we propose a more efficient algorithm to find the shortest valid path; this we call the SVP (Shortest Valid Path) algorithm.

IV. SHORTEST VALID PATH FOR SERIAL JOBS

A. The SVP algorithm

In order to tackle the problem highlighted in the application of Dijkstra’s algorithm, the following dismiss strategy is adopted by the SVP algorithm:

SVP records all jobs that an examined sub-path contains. Assume a set of sub-paths, \( S \), each of which contains the same set of jobs (the set of graph nodes that these paths traverse are different). SVP only keeps the path with the smallest distance and other paths are dismissed in further searches for the shortest path.

This strategy will clearly demonstrate improved efficiency compared with the intuitive, enumerative strategy, i.e., the SVP algorithm examines far fewer subpaths than the enumerative strategy. This is because, for all different subpaths that contain
Algorithm 1: The SVP Algorithm

1: SVP (Graph)
2: \( v.jjobset = \{ Graph.start \} \); \( v.path = Graph.start \);
3: \( v.distance = 0 \); \( v.level = 0 \);
4: Obtain \( v \) from \( Q \);
5: while Graph.end is not in \( v.jjobset \)
6: for every level \( l \) from \( v.level + 1 \) to
7: Graph.end.level do
8: if job \( i \) is not in \( v.jjobset \)
9: valid \( l \) = 1;
10: break;
11: \( k = l \);
12: while \( k \leq \binom{n-1}{u-1} \)
13: if \( node_{v jobset} \cap v.jjobset = \phi \)
14: \( J = v.jjobset \cup node_{v jobset} \);
15: \( J \) is not in \( Q \); Create an object \( u \) for \( J \);
16: \( u.jjobset = J \);
17: \( u.distance = \) distance;
18: \( u.path = v.path + node_{k} \);
19: \( u.level = node_{k}.level \);
20: Add \( u \) into \( Q \);
21: else
22: Obtain \( u' \) whose \( u'.jjobset \) is \( J \);
23: if \( distance < u'.distance \)
24: \( u'.distance = distance \);
25: \( u'.path = v.path + node_{k} \);
26: \( u'.level = node_{k}.level \);
27: \( k+1 \);
28: Remove \( v \) from \( Q \);
29: Obtain the \( v \) with smallest \( v.distance \) from \( Q \);
30: return \( v.path \) as the shortest valid path;

The same set of jobs, only one subpath (the shortest) will spawn further subpaths and all other subpaths will be discarded.

The SVP algorithm is outlined in Algorithm 1. The main differences between SVP and Dijkstra’s algorithm lie in three aspects: 1) The invalid paths, which contain the duplicate jobs, are disregarded by SVP during the searching; 2) The dismiss strategy is implemented; 3) No edges are generated between nodes before SVP starts and the node connections are established as SVP progresses. In this way, only the node connections spawned by the recorded subpaths will be generated and this will thereby further improve performance.

The time complexity of Algorithm 1 is \( O(\sum_{i=1}^{m} \binom{n-i}{u-1} \cdot (n-u+1) + \frac{n}{u} + \log(n)) \), where \( m \) is the number of u-core machines required to run \( u \) jobs. The detailed analysis of the time complexity is presented in the supplementary notes.

B. Further optimization of SVP

One of the most time-consuming steps in Algorithm 1 is to scan every node in a valid level to find a valid node for a given subpath \( v.path \) (Line 11 and 28). Theorem 1 is introduced to reduce the time spent in finding a valid node in a valid level. The rational behind Theorem 1 is that once the algorithm locates a node that contains a job appearing in \( v.path \), the number of nodes that follow that node and also contains that job can be calculated, since the nodes are arranged in ascending order of node ID. These nodes are all invalid and can therefore be ignored by the algorithm.

Theorem 1. Given a subpath \( v.path \), assume that level \( l \) is a valid level and node \( k \) (assume node \( k \) contains the jobs, \( j_{1}, \ldots, j_{u} \)) is the first node that is found to contain a job (assume the job is \( j_{i} \)) appearing in \( v.path \). Then, job \( j_{i} \) must also appear in the next \( \binom{n-j_{i}}{u-1} \) nodes at that level.

Proof: Since the graph nodes at a level are arranged in ascending order of node ID, the number of nodes whose \( i \)-th job is \( j_{i} \) equals the number of possibilities of mapping the jobs whose IDs are greater than \( j_{i} \) to \( (u-i) \) positions, which can be calculated by \( \binom{n-j_{i}}{u-1} \).

Based on Theorem 1, the O-SVP (Optimal SVP) algorithm is proposed to further optimize SVP. The only difference between O-SVP and SVP is that in the O-SVP algorithm, when the algorithm gets to an invalid node, instead of moving to the next node, it calculates the number of nodes that can be skipped and jumps to a valid node. Effectively, O-SVP can find a valid node in the time \( O(1) \). Therefore, the time complexity of O-SVP is \( O(\sum_{i=1}^{m} \binom{n-i}{u-1} \cdot (n-u+1)+ \log(n)) \)). The outline algorithm for O-SVP is omitted in this paper.

In summary, SVP accelerates the solving process over the enumerative method by reducing the length of Q in the algorithm while O-SVP further accelerates SVP by reducing the time spent in finding a valid node in a level.

V. Shortest Valid Path for Parallel Jobs

The SVP algorithm presented in the last section considers only serial jobs. This section addresses the co-scheduling of both serial and parallel jobs. Subsection V-A presents how to handle embarrassingly parallel (PE) jobs, while Subsection V-B further extends the work in Subsection V-A to handle parallel jobs with inter-process communications (PC).

A. Co-scheduling PE jobs

In Subsection V-A1, the SVPPE (SVP for PE) algorithm is proposed, extending SVP to incorporate PE jobs. Subsection V-A2 presents the optimization techniques used to accelerate the solving of SVPPE.

1) The SVPPE algorithm: When Algorithm 1 finds a valid node, it calculates the new distance after the current path extends to that node (Line 13). This calculation is adequate for serial jobs, but cannot be applied to parallel jobs. As discussed in Subsection III-B, the completion time of a parallel job is determined by Eq. 5. In order to incorporate parallel jobs, we can treat each process of a parallel job as a serial job (therefore the graph model remains the same) and extend the SVP algorithm simply by changing the means by which we calculate the path distance. In order to calculate the performance degradation for PE jobs, a number of new attributes are introduced. First, two new attributes are added to an object \( v \) in \( Q \). One attribute stores the total degradation of all serial jobs on \( v.path \) (denoted by \( v.dg_serial \)). The other attribute is an array, in which each entry stores the largest degradation of all processes of a parallel job \( p \) on \( v.path \) (denoted by \( v.dg_{p} \)). Second, two similar new attributes are also added to a graph node \( node_{k} \). One stores the total degradation of all serial jobs in \( node_{k} \) (denoted by \( node_{k}.dg_serial \)). The other is also an array, in which each entry stores the degradation of a parallel job \( p \) in \( node_{k} \) (denoted by \( node_{k}.dg_{p} \)).

SVPPPE is outlined in Algorithm 2. The only differences between SVPPE and SVP are: 1) Changing the means by which we calculate the subpath distance (Line 13-19 in Algorithm
The worst-case complexity of Algorithm 2 is therefore, combined with the time complexity of Algorithm 1, and updating the attributes (the two other for-loops) are which adjusts SVPPE so that it can find the shortest valid

2, and 2) Updating the newly introduced attributes for the case where J is not in Q (Line 28-30) and the case otherwise (Line 38-40).

The maximum number of iterations of all for-loops (Line 14, 28 and 38) is u, because there are at most u jobs in a node. Each iteration takes constant time. Therefore, the worst-case complexity of computing the degradation (the first for-loop) and updating the attributes (the two other for-loops) are O(u). Therefore, combined with the time complexity of Algorithm 1, the worst-case complexity of Algorithm 2 is O\left(\sum_{i=1}^{u} \left\lfloor\frac{n}{u+1}\right\rfloor + \log(n)\right).

2) Process condensation for optimizing SVPPE: An obvious optimization measure for SVPPE is to skip the invalid nodes in a similar way to that given in Theorem 1, which is not repeated in this Subsection. This subsection focuses on proposing another important optimization technique that is only applicable to PE jobs. The optimization technique is based on the following observation: different processes of a parallel job should have the same mutual effect with other jobs. So it is unnecessary to differentiate different processes of a parallel job, treating them as individual serial jobs.

Therefore, the optimization technique, which is called the process condensation technique in this paper, labels a process of a parallel job using its job ID, that is, it treats different processes of a parallel job as the same serial job. We illustrate this below using Figure 1. Now assume the jobs labelled 1, 2, 3 and 4 are four processes of a parallel job, whose ID is set to be 1. Figure 1 can be transformed to Figure 2 after deleting the same graph nodes in each level (the edges are omitted). Compared with Figure 1, it can be seen that the number of graph nodes in Figure 2 is reduced. Therefore, the number of subpaths that need to be examined and consequently the time spent in finding the optimal solution is significantly reduced.

\[ (n - u + 1) + u - \left\lfloor\frac{n}{u+1}\right\rfloor + \log(n) \]

\[ \left(\sum_{i=1}^{u} \left\lfloor\frac{n}{u+1}\right\rfloor + \log(n)\right) \]

\[ ((n - u + 1) + u - \left\lfloor\frac{n}{u+1}\right\rfloor + \log(n)) \]

\[ 2) \]

\[ \text{Theorem 2. Assume job } l \text{ is a parallel job. For a given subpath } v.path, \text{ level } l (l \text{ starts from } v.level + 1) \text{ is a valid level if } v.proc \text{ < } proc. \text{ Otherwise, level } l \text{ is not a valid level.} \]

\[ \text{Proof: Assume the jobs are co-scheduled on } u \text{-core machines. Let } U \text{ be the bag of jobs that includes all serial jobs and parallel jobs (the number of instances of a parallel job in } U \text{ equals the number of processes that that a job has). Let } D = U - v.jobset. X \text{ denotes all possible combinations of selecting } u - 1 \text{ jobs from } D. \text{ Because of the way that the nodes are organized in the graph, the last } u - 1 \text{ jobs of the nodes in level } l \text{ must include all possible combinations of selecting } u - 1 \text{ jobs from a set of jobs whose ID are in the range of } l \text{ to } n (n \text{ is the number of jobs to be co-scheduled), which is denoted by } Y. \text{ Then we must have } X \cap Y \neq \emptyset. \text{ This means that as long as the ID of the first job in the nodes in level } l \text{ is not making the nodes invalid, which can be determined by the condition } v.proc < proc, \text{ we must be able to find a node in level } l \text{ that can append to } v.path \text{ and form a new valid subpath.} \]

\[ \text{After a valid level is found, O-SVPPE needs to find a valid node in that level. When there are both parallel and serial jobs, O-SVPPE uses two conditions to determine a valid node: 1) the serial jobs in the node do not appear in } v.jobset, \text{ and 2) } \forall \text{ parallel job } p_i \text{ in the node, } v.proc + \text{ node}_k.proc \leq proc. \text{ O-SVPPE is outlined in Algorithm 3, in which Lines 7-13 implement the way of finding a valid level and Line 16 checks whether a node is valid, as discussed above.} \]

\[ B. \text{ Co-scheduling PC jobs} \]

We now extend the SVPPE algorithm to handle PC jobs, which is called SVPPC (SVP for PC jobs). We first model the communication time, c_{ij,S}, in Eq. 5 and then adjust SVPPE to handle PC jobs. Moreover, since the further optimization

Fig. 2: The graph model for a mix of serial and parallel jobs

We now present the O-SVPPE (Optimal SVPPE) algorithm, which adjusts SVPPE so that it can find the shortest valid
technique developed for PE jobs, i.e., the O-SVPPE algorithm, presented in Subsection V-A2 cannot be directly applied to PC jobs. The O-SVPPE algorithm is extended to handle PC jobs in Subsection V-B2, and termed O-SVPPC.

1) Modelling communications in PC jobs: $c_{ij,S}$ can be modelled using Eq. 6, where $\gamma_{ij}$ is the number of neighbouring processes that process $p_{ij}$ has, corresponding to the decomposition performed on the data set to be calculated by the parallel job; $\gamma_{ij}(k)$ is the amount of data that $p_{ij}$ needs to communicate with its $k$-th neighbouring process; $B$ is the bandwidth for inter-processor communications; $b_{ij}(k)$ is $p_{ij}$'s $k$-th neighbouring process, and $\beta_{ij}(k,S)$ is 0 or 1 as defined in Eq. 6b. $\beta_{ij}(k,S)$ is 0 if $b_{ij}(k)$ is in the job set S co-running with $p_{ij}$. Otherwise, $\beta_{ij}(k,S)$ is 1. Essentially, Eq. 6 calculates the total amount of data that $p_{ij}$ needs to communicate, which is then divided by the bandwidth $B$ to obtain the communication time. Note that $p_{ij}$'s communication time can be determined by only examining which neighbouring processes are not in the job set S co-running with $p_{ij}$, no matter which machines that these neighbouring processes are scheduled to. In the supplementary file to this paper, an example is given to illustrate the calculation of $c_{ij,S}$.

$$c_{ij,S} = \frac{1}{B} \sum_{k=1}^{n} (\gamma_{ij}(k) \cdot \beta_{ij}(k,S)) \quad (6a)$$

$$\beta_{ij}(k,S) = \begin{cases} 
0 & \text{if } b_{ij}(k) \in S \\
1 & \text{if } b_{ij}(k) \notin S 
\end{cases} \quad (6b)$$

We now adjust SVPPE to incorporate the PC jobs. In the graph model for serial and PE jobs, the weight of a graph node is calculated by summing up the weights of the individual jobs/processes, which is the performance degradation. When there are PC jobs, a process belongs to a PC job, the weight of a process $p_{ij}$ in a PC job should be calculated by Eq. 5 instead of Eq. 1. The remainder of the SVPPC algorithm is exactly the same as SVPPE.

2) Communication-aware process condensation for optimizing SVPPC: The reason why the process condensation technique developed for PE jobs cannot be directly applied to PC jobs is because different processes in a PC job may have different communication patterns and therefore cannot be treated as identical processes. After carefully examining the characteristics of the typical inter-process communication patterns, a communication-aware process condensation technique is developed to accelerate the solving process of SVPPC, which is called O-SVPPC (Optimized SVPPC) in this paper.

We can construct the co-scheduling graph model as we did in Fig. 1 for finding the optimal solution of co-scheduling PC and serial jobs. We then define the communication property of a parallel job in a graph node as the number of communications that the processes of the parallel job in the graph node has to perform in each decomposition direction with other nodes. In the communication-aware process condensation, multiple graph nodes in the same level of the graph model can be condensed to one node if the following two conditions are met: 1) these nodes contain the same set of serial jobs and parallel jobs, and 2) the communication properties of all PCs in these nodes are the same. A concrete example is presented in the supplementary file to illustrate the condensation process.

VI. CO-SCHEDULING JOBS ON MULTI-PROCESSOR COMPUTERS

In order to add more cores to a multicore computer, there are two general approaches: 1) increase the number of cores on a processor chip and 2) install more processors, with the number of cores in each processor remaining unchanged; both approaches are often simultaneously applied.

The co-scheduling graph previously presented is for multi-core machines each of which contains a single multi-core processor, which we term a single processor multicore machine (or a single processor for short). If there are multiple multi-core processors in a machine (which we term a multi-processor machine), the resource contention, such as cache contention, is different. For example, only the cores on the same processor share the Last-Level Cache (LLC) on the chip, while the cores on different processors do not compete for cache. In a single processor machine, the job-to-core mapping does not affect the tasks’ performance degradation. This is not the case in a multi-processor machine, as illustrated in the following example.

Consider a machine with two dual-core processors (processors $p_1$ and $p_2$) and a co-run group with 4 jobs ($j_1, ..., j_4$). Now consider two job-to-core mappings. In the first mapping, jobs $j_1$ and $j_2$ are scheduled on processor $p_1$, while $j_3$ and $j_4$ are scheduled on $p_2$. In the second mapping, jobs $j_1$ and $j_3$ are scheduled on processor $p_1$, while $j_2$ and $j_4$ are scheduled on $p_2$. The two mappings generate different total performance degradations for this co-run group. In the co-scheduling graph in previous sections, a graph node corresponds to a possible co-run group in a machine, which is associated with a single performance degradation value. This holds for a single processor machine. As shown in the above discussions, however, a co-run group may generate different performance degradations in a multi-processor machine, depending on the job-to-core mapping within the machine. This subsection presents how to adjust the methods presented in previous sections to find the optimal co-scheduling solution in multi-processor machines.

A straightforward method is to generate multiple nodes in the co-scheduling graph for a possible co-run group, with each node having a different weight that equals a different performance degradation value (which is determined by the specific job-to-core mappings). We call this method MNG (Multi-Node for a co-run Group) method. For a machine with $p$ processors with each processor having $u$ cores, it can be calculated that $\frac{p}{u+1} \prod_{i=0}^{u} \binom{p-1}{u} \frac{u}{p}$ different job-to-core mappings that may produce different performance degradations. The algorithms presented in previous sections can be used to find the shortest path in this co-scheduling graph, where the shortest path must
correspond to the optimal co-scheduling solution on the multi-
processor machines. In this straightforward solution, however,
the scale of the co-scheduling graph (i.e., the number of
graph nodes) increases \( \prod_{k=1}^{n-1} \binom{p-k}{p} \) fold, and consequently the
solving time increases significantly compared with that for the
case of single processor machines.

We now propose a method, called the Least Performance
Degradation (LPD) method, to construct the new
co-scheduling graph. Using this method, the optimal co-
scheduling solution for multi-processor machines can be com-
puted without increasing the scale of the co-scheduling graph.
The LPD method is explained below.

As discussed above, in the case of multi-processor ma-
chines, a co-run group may produce different performance
degradation in a multi-processor machine. Instead of gener-
ating multiple nodes (each being associated with a different
weight, i.e., a different performance degradation value) in the
co-scheduling graph for a co-run group, the LPD method
constructs the co-scheduling graph for multi-processor ma-
chines in the following way: A node is generated for a co-run
group and the weight of the node is set to be the smallest
performance degradation among all possible performance
degradations generated by the co-run group. The remainder
of the construction process is exactly the same as that for the
case of single processor machines.

Theorem 3 proves that from the co-scheduling graph con-
structed by the LPD method, the algorithms proposed in pre-
vious sections for the case of single processor machines still
obtain the optimal co-scheduling solution on multi-processor
machines.

Theorem 3. Assume the jobs are to be co-scheduled on multi-
processor machines. Using the LPD method defined above to
construct the co-scheduling graph, the algorithms that have
been proposed to find the optimal co-scheduling solutions on
single processor machines will still find the optimal co-
scheduling solutions on multi-processor machines.

Proof: We can use the MNG method or the LPD method
to construct the co-scheduling graph for the case of multi-
processor machines. It has been discussed above that when
using the MNG method to construct the graph, the algorithms
proposed for single processor machines can still find the
optimal co-scheduling solution on multi-processor machines.
In the co-scheduling graph constructed by the MNG method,
multiple nodes are created for a possible co-run group, each
with a different weight. If a co-run group appears in the
final shortest path obtained by the algorithms, the path must
only contain the node with the least weight for the co-run
group. Other nodes with higher weights would have been
dismissed in the process of searching for the shortest path.
Therefore, the shortest path obtained from the co-scheduling
graph constructed by the LPD method must be the same as
that from the graph by the LPD method. Consequently, the
theorem holds.

VII. CO-SCHEDULING MULTI-THREAD JOBS

A parallel job considered so far in this paper is one
consisting of multiple processes, such as an MPI job. In this
subsection, we adapt the proposed graph model and algorithms
so that they can handle parallel jobs consisting of multiple
threads, such as OpenMP jobs. We call the former parallel
jobs Multi-Process Parallel (MPP) jobs and the latter Multi-
Thread Parallel (MTP) jobs.

In the co-scheduling graph, a thread in an MTP job is
treated in the same way as a parallel process in an MPP
job. Compared with MPP jobs, however, MTP jobs have
the following different characteristics: 1) multiple threads
of a MTP job must reside in the same node, and 2) the
communication time between threads can be largely ignored.
Accordingly, the co-scheduling graph model is adjusted as
follows to handle the MTP jobs. For each node (i.e., every
possible co-run group) in the co-scheduling graph, we check
whether all threads belonging to the MTP are on the node. If
not, the node is deleted from the graph since it does not satisfy
the condition that all threads of a MTP job must reside in the
same node. We call the above process the validity check for
MTP jobs.

Since the communication time between the threads in MTP
jobs can be ignored, the performance degradation of a MTP
job can be calculated using Eq. 3 that is used to compute
the performance degradation of a PE job. Also, since the
communication time of an MTP job is not considered, an
intuitive method to find the optimal co-scheduling solution in
the presence of MTP jobs is to use the algorithm for handling
PE jobs, i.e., Algorithm 3. However, after closer inspection
into the features of MTP jobs, it is apparent that Algorithm
3 can be adjusted to improve the performance of managing
MTP jobs, a feature which is explained next.

After the validity check for MTP jobs, all threads belonging
to a MTP job must only appear in the same graph node. There-
fore, there is no need to perform the process condensation as
we do in the presence of PE jobs. Consequently, the SVPPE
algorithm (i.e., Algorithm 2) can be used to handle MTP jobs.
Next, when the current path expands to a new node in the
SVPPE Algorithm, for each parallel job \( p_i \) in the new node,
SVPPE needs to check whether \( p_i \) appears in the current path.
However, all threads in a MTP job only reside in the same
node. Therefore, if a new node that the current path tries to
expand to contains an MTP job, it is unnecessary to check
whether threads of the MTP job appear in the current path.

In order to differentiate this approach from SVPPE, the
algorithm for finding the optimal co-scheduling solution for
the mix of serial and MTP jobs is denoted as SVPPPT (where
T stands for thread). The only difference between SVPPPT and
SVPE is that Lines 15-17 in SVPPPE (i.e., Algorithm 2) are
removed from SVPPPT.

From the above discussions, we can conclude that it is more
efficient to find the optimal co-scheduling solution for MTP
jobs than for PE jobs. This is because 1) the number of nodes
in the co-scheduling graph for SVPPPT is much less than that
for PE jobs (because of the validity check for MTP jobs) and
2) SVPPPT does not execute Lines 15-17 in SVPPPE.

Note that the method described above for handling MTP
jobs is applicable to both single processor machines and multi-
processor machines, as defined previously.

VIII. THE A*-SEARCH-BASED ALGORITHM

The dismiss strategy designed for the SVP algorithm in
Subsection IV-A and the optimization strategies developed in
O-SVPPE and O-SVPPC can avoid unnecessary searches in
the co-scheduling graph. It has been shown that the A*-search
algorithm is also able to find the optimal solution and during the searching, effectively prune the graph branches that will not lead to the optimal solution. In order to further accelerate the solving process, an A*-search-based algorithm is developed in this section to combine the ability of avoiding the unnecessary searches in the traditional A*-search algorithm and the algorithms presented in this paper so far (SVP, O-SVP, O-SVPPE and O-SVPPC).

This section presents how to design the A*-search-based algorithm to find the optimal co-scheduling solution in the co-scheduling graph. We only consider the co-scheduling of serial and PC jobs for the sake of generality. The presented A*-search-based algorithm is called SVPPC-A*. SVP-A* (i.e., co-scheduling serial jobs), SVPPE-A* (i.e., co-scheduling both serial and PE jobs) and SVPPPT-A* can be developed in a similar way.

The traditional A*-search algorithm, which is briefly overviewed in the supplementary notes, cannot be directly applied to obtain the optimal co-scheduling solution for the same reasons discussed in the presentation of the SVP and the SVPPC algorithms; namely, i) the optimal co-scheduling solution in the constructed co-scheduling graph corresponds to the shortest valid path, not the shortest path, and ii) since the jobs to be scheduled contain parallel jobs, the distance of a path is not the total weights of the nodes on the path, as calculated by the traditional A*-search algorithm.

Three functions are defined in the traditional A*-search algorithm. Function $g(v)$ is the actual distance from the start node to node $v$ and $h(v)$ is the estimated length from $v$ to the end node, while $f(v)$ is the sum of $g(v)$ and $h(v)$. In SVPPC-A*, we use the exactly same methods proposed for the SVP algorithm (i.e., the dismiss strategy) to handle and expand the valid subpaths and avoid the unnecessary searches. Also, we use the method proposed for the SVP algorithm to calculate the distance of the subpaths (i.e., Eq. 3 and Eq. 5) that contain the PE jobs. This technique can be used to obtain the value of $g(v)$. Note that the communication-aware process condensation technique proposed in Subsection V-B2 can also be used to accelerate SVPPC-A*.

The estimation of $h(v)$ is one of the most critical parts in designing an A*-search algorithm. Function $g(v)$ is the actual distance from the start node to node $v$ and $h(v)$ is the estimated length from $v$ to the end node, while $f(v)$ is the sum of $g(v)$ and $h(v)$. In SVPPC-A*, we use the exactly same methods proposed for the SVP algorithm (i.e., the dismiss strategy) to handle and expand the valid subpaths and avoid the unnecessary searches. Also, we use the method proposed for the SVP algorithm to calculate the distance of the subpaths (i.e., Eq. 3 and Eq. 5) that contain the PE jobs. This technique can be used to obtain the value of $g(v)$. Note that the communication-aware process condensation technique proposed in Subsection V-B2 can also be used to accelerate SVPPC-A*.

The estimation of $h(v)$ is one of the most critical parts in designing an A*-search algorithm. The following two properties reflect the importance of $h(v)$ [18]: i) The result of an A* search is optimal if the estimation of $h(v)$ is not higher than the lowest cost to reach the end node, and ii) the closer the result of $h(v)$ is from the lowest cost, the more effective A* search is in pruning the search space.

Therefore, in order to find the optimal solution, the $h(v)$ function must satisfy the first property. In our problem, if there are $q$ jobs on the path corresponding to $g(v)$, then the aim of setting the $h(v)$ function is to find a function of the remaining $n - q$ jobs such as the value of the function is less than the shortest distance from node $v$ to the end node. The following two strategies are proposed to set the $h(v)$ function.

\textbf{Strategy 1 for setting $h(v)$:} Assume node $v$ is in level $l$, we construct a set $R$ that contains all the nodes from $l + 1$ to the last level in the co-scheduling graph, and sort these nodes in ascending order of their weights. Then, regardless of the validity, the first $(n - q)/u$ (where $u$ is the number of cores) nodes are selected from $R$ to form a new subpath; the distance of this subpath is $h(v)$.

\textbf{Strategy 2 for setting $h(v)$:} Assume node $v$ is in level $l$. We find all valid levels from level $l + 1$ to the last level in the co-scheduling graph. The total number of valid levels obtained must be $(n - q)/u$. We then obtain the node with the least weight from each valid level. $(n - q)/u$ nodes will be obtained. We use these $(n - q)/u$ nodes to form a new subpath and use its distance as $h(v)$.

It is easy to prove that $h(v)$, obtained through the above strategies, must be less than the actual shortest distance from $v$ to the end node; this is the case because it uses the nodes with the smallest weights from all remaining nodes in Strategy 1 or from all valid levels in Strategy 2. We will show in the experiments that Strategy 2 is much more effective than Strategy 1 in terms of pruning unnecessary searches.

\section{IX. Clustering approximation for finding the shortest valid path}

We have presented methods and optimization strategies for solving the graph model for the shortest valid path. In order to further shorten the solving time and strike a balance between solving efficiency and solution quality, this section proposes a flexible technique called the clustering technique, to rapidly find an approximate solution. The clustering technique is flexible because the solving efficiency can be adjusted by setting the desired solution quality. It can be applied to O-SVP, O-SVPPE and O-SVPPC.

As discussed in introduction and related work, the reason why co-scheduling causes performance degradation is because the co-running jobs compete for shared cache. SDC (Stack Distance Competition) is a popular technique for calculating the impact when multiple jobs are co-running; this uses the SDPs (Stack Distance Profile) of the multiple jobs as input. Therefore, if two jobs have similar SDPs, they will have a similar effect on other co-running jobs. The fundamental idea of the proposed clustering technique is to class the jobs with similar SDPs together and treat them as the same job. Reflected in the graph model, the jobs in the same class can be given the same job ID. In so doing, the number of different nodes in the graph model will be significantly reduced. The resulting effect is the same as when different parallel processes are given the same job ID in the O-SVPPE algorithm in Subsection V-A2.

We now introduce a method of measuring the SDP similarity between two jobs. Given a job $j_i$, its SDP is essentially an array, in which the $k$-th element records the number of cache hits on the $k$-th cache line (which is denoted by $h_i[k]$). The following formula is used to calculate the Similarity Level (SL) in terms of SDP when comparing another job $j_j$ against $j_i$.

$$SL = \frac{\sqrt{\sum_{k=1}^{cl} (h_i[k] - h_j[k])^2}}{\sum_{k=1}^{cl} h_i[k]}$$

When SL is larger, more jobs will be classed together. Consequently, there will be fewer nodes in the graph model and hence less scheduling time is needed to calculate an accurate solution.

The O-SVP clustering algorithm is the same as the O-SVP algorithm except in the way a valid level as well as a valid node in a valid level is found, which is the same as that for O-SVPPE (Algorithm 3). The clustering technique can also be applied to O-SVPPE and O-SVPPC in a similar way. Detailed discussion of this process is not repeated.
X. Evaluation

This section evaluates the effectiveness and the efficiency of the proposed methods: O-SVP, O-SVPPE, O-SVPPC, A*-search-based algorithms (i.e., SVPPC-A* and SVP-A*) and the clustering approximation technique. In order to carry out this evaluation, we compare the algorithms proposed in this paper with existing co-scheduling algorithms proposed in [18]: Integer Programming (IP), Hierarchical Perfect Matching (HPM), and Greedy (GR).

We conduct the experiments with real jobs. Serial jobs are taken from the NASA benchmark suit NPB3.3-SER [4] and SPEC CPU 2000 [5]. NPB3.3-SER has 10 serial programs and each program has 5 different problem sizes. The problem size used in the experiments is size C. The PC jobs are selected from the ten MPI applications in the NPB3.3-MPI benchmark suite. As for PE jobs, 5 embarrassingly parallel programs are used: PI [3], Mandelbrot Set(MMS) [2], RandomAccess(RA) from the HPCC benchmark [1], EP from NPB-MPI [4] and Markov Chain Monte Carlo for Bayesian inference (MCM) [21]. In all these 5 embarrassingly parallel programs, multiple slave processes are used to perform calculations in parallel and a master process reduces the final result after it gathers the results from all slaves. This set of parallel programs is selected because they contains both computation-intensive (e.g. MMS and PI) and memory-intensive programs (e.g. RA).

Four types of machines, Dual core, Quad core, 8 core and 16 core machines, are used to run the benchmarking programs. The dual-core machine has an Intel Core 2 Dual processor and each core has a dedicated 32KB L1 data cache and a 4MB 16-way L2 cache shared by the two cores. The Quad-core machine has an Intel Core i7 2600 processor and each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache. A further 8MB 16-way L3 cache is shared by the four cores. The 8 core machine has two Intel Xeon L5520 processors with each processor having 4 cores. Each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache, and an 8MB 16-way L3 cache shared by 4 cores. The 16 core machine has two Intel Xeon E5-2450L processors with each processor having 8 cores. Each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache, and a 16-way 20MB L3 cache shared by 8 cores. The network interconnecting the dual-core and quad-core machines is a 10 Gigabit Ethernet, while the network interconnecting the 8-core and 16-core Xeon machines is QLogic TrueScale 4X QDR InfiniBand. In the remainder of this section, we label the 8 core and 16 core machines as 2*4 core and 2*8 core machines, to highlight the fact that they are dual-processor machines.

The single-run computation times of the benchmarking programs are measured. Then the method presented in [25] is used to estimate the co-running computation times of the programs, the details of which are presented in the supplementary notes. With the single-run and co-run computation times, Eq. 1 is then used to compute the performance degradation.

In order to obtain the communication time of a parallel process when it is scheduled to co-run with a set of jobs/processes, i.e., $c_{ij,s}$ in Eq. 6, we examined the source codes of the benchmarking MPI programs used for the experiments and obtained the amount of data that the process needs to communicate with each of its neighbouring processes (i.e., $\alpha_{ij}(k)$ in Eq. 6). Then Eq. 6 is used to calculate $c_{ij,s}$.

A. Evaluating the O-SVP algorithm

In this subsection, we compare the O-SVP algorithm with the existing co-scheduling algorithms in [18].

These experiments use all 10 serial benchmark programs from the NPB-SER suite. The results are presented in 3a and 3b, which show the performance degradation of each of the 10 programs plus their average degradation under different co-scheduling strategies on Dual-core and Quad-core machines.

The work in [18] shows that IP generates the optimal co-scheduling solutions for serial jobs. As can be seen from Figure 3a, O-SVP achieves the same average degradation as that under IP. This suggests that O-SVP can find the optimal co-scheduling solution for serial jobs. The average degradation produced by GR is 15.2% worse than that of the optimal solution. It can also been seen from Figure 3a that the degradation of FT is the largest among all 10 benchmark programs. This may be because FT is the most memory-intensive program among all those selected, and therefore endures the largest degradation when it has to share the cache with competing jobs.

Figure 3b shows the results on Quad-core machines. In this experiment, in addition to the 10 programs from NPB-SER, 6 serial programs (applu, art, ammp, equake, galgel and vpr) are selected from SPEC CPU 2000. In Figure 3b, O-SVP produces the same solution as IP, which shows the optimality of O-SVP. O-SVP also finds a better co-scheduling solution than HPM and GR. The degradation under HPM is 7.7% worse than that under O-SVP, while GR is 25.5% worse. It is worth noting that O-SVP does not produce the least degradation for all programs. The aim of O-SVP is to produce minimal total degradation. Hence, O-SVP produced a larger degradation than GR and HPM in some cases.

B. The O-SVPPE algorithm

We propose O-SVPPE because 1) none of the existing co-scheduling methods are designed for parallel jobs; 2) we argue that applying existing co-scheduling methods designed for serial jobs to scheduling parallel jobs will not produce optimal solutions. In order to investigate the performance discrepancy between the methods for serial jobs and PE jobs, we apply O-SVP to solve the co-scheduling for a mix of serial and parallel jobs and compare the results with those obtained by O-SVPPE. The mix of serial and parallel jobs consist of 5 embarrassingly parallel programs (each with 12 processes) and serial jobs from NPB-SER plus art from SPEC CPU 2000. Experimental results are shown in Figure 4a and 4b.

As can be seen from the figures, SVPPE produces smaller average degradation than O-SVP in both Dual-core and Quad-core cases. In the Dual-core case, the degradation under O-SVP is worse than that under SVPPE by 9.4%, while in the Quad-core case, O-SVP is worse by 35.6%. These results suggest that it is necessary to design co-scheduling methods for parallel jobs.

C. The O-SVPPC algorithm

Figure 5a and 5b show the Communication-Combined Degradation (CCD) (i.e., the value of Eq. 5) of the co-scheduling solution obtained by the SVPPC algorithm when the applications are run on Dual-core and Quad-core machines, respectively. In this set of experiments, 5 MPI applications
We further investigate the impact on CCD as the number of parallel jobs or parallel processes increases. Experimental results are shown in Figure 6a and 6b (on Quad-core machines). In Figure 6a, the number of total jobs/processes is 64. The number of parallel jobs is 4 (LU-Par, MG-Par, SP-Par and CG-Par) and the number of processes per job increases from 12 to 16; all other jobs are serial. For example, 8+4*12 represents a job mix with 8 serial and 4 parallel jobs, each with 12 processes.

In Figure 6a, the difference in CCD between SVPPC and SVPPE becomes larger as the number of parallel processes increases. This result suggests that SVPPE performs increasingly worse than SVPPC (increasing from 11.8% to 21.5%) as the proportion of PC jobs increases in the job mix. Another observation from this figure is that the CCD decreases as the proportion of parallel jobs increases. This is simply because the degradation experienced by multiple processes of a parallel job is only counted once. If those processes are serial jobs, their degradations will be summed and is therefore larger. In Figure 6b, the number of processes per parallel job remains unchanged and the number of parallel jobs increases. For example, 12+2*4 represents a job mix with 12 serial jobs and 2 parallel jobs, each with 4 processes. The detailed combinations of serial and parallel jobs are: i) In the case of 16+1*4, MG-Par is used as the parallel job and all 16 serial programs are used as the serial jobs; ii) In the case of 12+2*4, LU-Par and MG-Par represent parallel jobs and the serial jobs are SP, BT, FT, CG, IS, UA, applu, art, ammp, equake, galgel and vpr; iii) In the case of 8+3*4, BT-Par, LU-Par, MG-Par represent the parallel jobs and the serial jobs are SP, BT, FT, DC, IS, UA, equake, galgel and vpr. The results in Figure 6b show a similar pattern to those in Figure 6a; similar reasoning holds.

D. Scheduling in Multi-processor Computers

In this section, we investigate the effectiveness of the LPD method proposed to handle co-scheduling in multi-processor machines. In the experiments, we first use the MNG method discussed in Section VI (i.e., generating multiple graph nodes for a co-run group, with each node having a different weight) to construct the co-scheduling graph. As we have discussed, from the co-scheduling graph constructed by...
the MNG method, the algorithm must be able to find the optimal co-scheduling solution for multi-processor machines. We then use the LPD method to construct the graph and find the shortest path in the graph. The experimental results are presented in Figure 7a and 7b, in which a mix of 4 PE jobs (PI, MMS, RA, and MCM, each with 31 processes) and 4 serial jobs (DC, UA, BT and IS) are used. It can be seen that the performance degradations obtained by the two methods are the same. This result verifies that the algorithms can produce optimal co-scheduling solutions using the LPD method.

Following the same logic as in Figure 4, we conduct experiments to investigate the performance discrepancy between the method for serial jobs and that for PE jobs on multi-processor machines. The LPD method is used to generate the co-scheduling graphs (therefore, the “LPD” prefix is added to the algorithms in the legends of the figures). In these experiments, we use the same experimental settings as in Figure 7a and 7b. The results are shown in Figure 8a and 8b. As can be seen from the figures, LPD-SVPPE produces smaller average degradation than LPD-SVP in both 8-core and 16-core cases. In the 8-core case, the degradation under LPD-SVP is worse than that under LPD-SVPPE by 31.9%, while in the 16-core case, LPD-SVP is worse by 34.8%. These results verify the effectiveness of the LPD method for co-scheduling PE jobs.

Similarly, following the same logic as in Figure 5, we conduct experiments to run PC jobs using SVPPC and SVPPE on multi-processor machines and compare the performance discrepancy in terms of CCD. The same experimental settings as in Figure 5 are used and the results are presented in Figure 9a and 9b. In this set of experiments, 4 MPI applications (BT-Par, LU-Par, MG-Par and CG-Par) are selected from the NPB3.3-MPI suite and each parallel application is run using 31 processes, while the same serial jobs as in Fig. 7a are used. As can be seen from these figures, the CCD under LPD-SVPPE is worse than that under LPD-SVPPC by 36.1% and 39.5% on 8-core and 16-core machines, respectively. These results justify using SVPPC to handle PC jobs and show that the LPD method works well with the SVPPC algorithm.

As discussed in Section VI, the reason why we propose the LPD method is because using the MNG method, the scale of the co-scheduling graph increased significantly in multi-processor systems. The LPD method can reduce the scale of the co-scheduling graph and consequently reduce the solving time. Therefore, we also conduct experiments to compare the solving time obtained by LPD and the MNG method. The experimental results are presented in Figure 10, in which Figure 10a and 10b are for PE and PC jobs, respectively. It can be seen from the figure that the solving time of LPD is significantly less than that of the straightforward method and the discrepancy increases dramatically as the number of jobs increases. These results suggest that LPD is effective in reducing the solve time compared with the MNG method.

E. Scheduling Multi-threaded Jobs

In Section VII, in order to schedule the MTP jobs correctly, we need to guarantee that the threads from the same MTP job are scheduled on the same node. In order to address this, the SVPPPT algorithm is proposed to construct the co-scheduling graph and find the shortest path. In this subsection, we first conduct experiments to examine the co-scheduling solution
Average Degradation
Quad Core
SVPPC-A*
O-SVPPC
SVP-A*
0.07
0.098
0.12
0.36
0.13
0.34
SVP-A*
0.15
0.34
Dual Core
0.22
0.74
Quad Core
0.098
O-SVP
0.27
SVPPC-A*
0.05

As can be seen from Table III and IV, SVP-A* and SVPPC-A* achieve the same performance degradation as those of O-SVP and O-SVPPC, respectively. These results verify the optimality of the A*-search-based algorithms. Indeed, SVPPC-A* combines the functionalities of SVPPC and the A*-search algorithm and is expected to generate the optimal solution.

Table V and VI show the scheduling efficiency of our A*-search-based algorithms under the two different strategies of setting the \( h(v) \) function proposed in Section VIII. SVP-A*-1 (or SVPPC-A*-1) and SVP-A*-2 (or SVPPC-A*-2) are the SVP-A* (or SVPPC-A*) algorithm that uses Strategy 1 and 2, respectively, to set \( h(v) \). Table V shows the results for synthetic serial jobs, while Table VI shows the results for parallel jobs. In Table VI, 4 synthetic parallel jobs are used and the number of processes of each parallel job increases from 10 to 50. Recall that the O-SVP is equivalent to SVPPC-A* with the \( h(v) \) function set to 0, while O-SVPPC is equivalent to SVPPC-A* with \( h(v) \) set to 0. Therefore, we also conduct experiments to show the scheduling efficiency of O-SVP and O-SVPPC, which can be used to demonstrate the effectiveness of the strategies of setting \( h(v) \). The underlying reason why SVPPC-A* and SVP-A* could be effective is because they can further avoid unnecessary searches in the constructed co-scheduling graph. Therefore, we also record the number of paths visited by each algorithm and present these in Tables V and VI.

It can be seen from both tables that the strategies used to set \( h(v) \) play a critical role in our A*-search-based algorithms. Both Strategy 1 and 2 proposed in Section VIII can reduce the number of visited paths dramatically and therefore reduce the solve time compared with the corresponding O-SVP and O-SVPPC algorithms. These results suggest that the strategies proposed in this paper can significantly reduce unnecessary searches.

Further, as observed from Table V and VI, the algorithms under Strategy 2 visited fewer paths by orders of magnitude than their counterparts under Strategy 1. Therefore, SVP-A*-2 and SVPPC-A*-2 are more efficient by orders of magnitude than SVP-A*-1 and SVPPC-A*-1, respectively, in finding the optimal co-scheduling solution. This is because the estimation of \( h(v) \) provided by Strategy 2 is much closer to the actual shortest path of the remaining nodes than that determined by Strategy 1, and consequently Strategy 2 is much more effective than Strategy 1 in avoiding unnecessary searches.

The scalability of the proposed algorithms can also be observed from Table V and VI. It can be seen that SVPPC-A*-2 (or SVP-A*-2) shows the best scalability against SVPPC-A*-1 and O-SVPPC (or SVP-A*-1 and O-SVP). This can be explained as follows: Although the scale of the constructed co-scheduling graph and the possible search paths increase rapidly as the number of jobs/processes increases, SVPPC-A*-2 can effectively prune the graph’s branches that will not lead to the optimal solution. Therefore, the increase in the scale of the graph will not increase the solve time of SVPPC-A*-2 as much as for other two algorithms.

**G. The optimization techniques**

We test the efficiency of the communication-aware process condensation techniques and the clustering approximation pro-
TABLE V: Comparison of the strategies for setting \( h(v) \) with serial jobs

<table>
<thead>
<tr>
<th>Number of Jobs</th>
<th>Solve time (seconds)</th>
<th>O-SVP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVP-A*-1</td>
<td>SVP-A*-2</td>
</tr>
<tr>
<td>16</td>
<td>0.72</td>
<td>0.014</td>
</tr>
<tr>
<td>20</td>
<td>12.88</td>
<td>0.047</td>
</tr>
<tr>
<td>24</td>
<td>190.79</td>
<td>0.14</td>
</tr>
</tbody>
</table>

TABLE VI: Comparison of the strategies for setting \( h(v) \) with parallel jobs

<table>
<thead>
<tr>
<th>Number of Processes</th>
<th>Solve time (seconds)</th>
<th>O-SVP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVPPC-A*-1</td>
<td>SVPPC-A*-2</td>
</tr>
<tr>
<td>40</td>
<td>0.43</td>
<td>0.037</td>
</tr>
<tr>
<td>80</td>
<td>2.44</td>
<td>0.17</td>
</tr>
<tr>
<td>120</td>
<td>10.93</td>
<td>0.33</td>
</tr>
<tr>
<td>160</td>
<td>40.05</td>
<td>0.64</td>
</tr>
<tr>
<td>200</td>
<td>99.13</td>
<td>0.88</td>
</tr>
</tbody>
</table>

TABLE VII: Comparing the clustering method with O-SVP

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>visited path</th>
<th>Degradation time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O-SVP</td>
<td>181267589</td>
<td>19.97</td>
</tr>
<tr>
<td>8 class</td>
<td>2115716</td>
<td>21.25</td>
</tr>
<tr>
<td>4 class</td>
<td>141508</td>
<td>23.75</td>
</tr>
<tr>
<td>2 class</td>
<td>17691</td>
<td>25.96</td>
</tr>
</tbody>
</table>

XI. CONCLUSIONS AND FUTURE WORK

In this research we propose a graph-based method to co-schedule jobs in multi-core computers. A graph is constructed for the co-scheduling problem, then finding the optimal co-scheduling solution is modelled as finding the shortest valid path in the graph. An algorithm for finding the shortest valid path for serial jobs is first developed and then an optimization strategy is proposed to reduce the solve time. Further, the algorithm for serial jobs is extended to incorporate parallel jobs; the optimization strategies are also developed to accelerate the solving process for finding the optimal solution for parallel jobs. Moreover, a flexible approximation technique is proposed to balance solving efficiency and solution quality. Experiments have been conducted to verify the effectiveness of the proposed algorithms. Future work is two-fold: 1) It is possible to parallelize the proposed co-scheduling algorithms to further speedup the process of finding the optimal solution. We plan to investigate a parallel paradigm suitable for this problem and design suitable parallelization strategies; 2) We plan to extend our co-scheduling methods to solve the optimal mapping of virtual machines (VM) on physical machines. The main extension necessary to the proposed methods is to allow VM migration between physical machines.

ACKNOWLEDGEMENTS

We acknowledge a number of different funding sources that support this work: The Royal Society Industry Fellowship Scheme (IF090020/AM); Bull Information Systems and the Bull/ Warwick Premier Partnership and the PhD funding that this provides; EPSRC and Intel Corporation Industrial CASE Account (awards 38405-2013 and 15220082-2015).

REFERENCES


Ligang He received Bachelors and Masters degrees from Huazhong University of Science and Technology, China, and a PhD in Computer Science from the University of Warwick, UK. He spent his early post-doctoral years as a researcher at the University of Cambridge, UK. In 2006, he joined the Department of Computer Science at the University of Warwick as an Assistant Professor and was subsequently promoted to Associate Professor. His interests include parallel and distributed computing, high performance computing and cloud computing.

He has published more than 70 papers in international conferences and journals, such as TPDS, JPDC, IPDPS, ICPP, ICWS, MASCOTS, etc. He has served as a member of the program committee or Conference Chair for a number of international conferences. He is a member of the IEEE.

Huanzhou Zhu received a Bachelors degree in Information System Engineering from Imperial College London, UK, and a Masters degree in Computer Science from the University of Warwick, UK. He is now a PhD student in Computer Science at the University of Warwick. His areas of interest include parallel and distributed computing, high performance computing and cloud computing.
Stephen Jarvis is Professor of High Performance Computing at the University of Warwick; he currently also serves as Chair of the Department of Computer Science at the University of Warwick, UK. While previously at the Oxford University Computing Laboratory, he worked on the development of performance tools with Oxford Parallel, Sychron Ltd., and Microsoft Research, Cambridge. He has close research ties with industry, including projects with IBM, BT, Hewlett-Packard Research Laboratories, Intel, Bull/ATOS, Rolls-Royce and AWE. He has authored more than 150 refereed publications (including three books) in the area of software and performance evaluation. He is a co-organizer for one of the UKs high-end scientific computing training centers and currently holds a William Penney Fellowship, recognising his advisory role to the UK MOD. He is a member of the IEEE.