Optimal Design of Multi-product Batch Plants Using a Parallel Branch-and-Bound Method

Andrey Borisenko[1], Philipp Kegel[2], and Sergei Gorlatch[2]

[1] Tambov State Technical University, Russia
borisenko@mail.gaps.tstu.ru
[2] University of Muenster, Germany
{philipp.kegel|gorlatch}@uni-muenster.de

Abstract. In this paper we develop and implement a parallel algorithm for a real-world application: finding optimal designs for multi-product batch plants. We describe two parallelization strategies – for systems with shared-memory and distributed-memory – based on the branch-and-bound paradigm and implement them using OpenMP (Open Multi-Processing) and MPI (Message Passing Interface), correspondingly. Experimental results demonstrate that our approach provides competitive speedup on modern clusters of multi-core processors.

Keywords: multi-product batch plant, parallel optimization, branch-and-bound, master-worker, global optimization, MPI, OpenMP

1 Motivation and Related Work

Selecting the equipment of a Chemical-Engineering System (CES) is one of the main problems when designing chemical multi-product batch plants, e.g., for synthesizing chemical dyes and intermediate products, photographic materials, pharmaceuticals etc. A solution of this problem comprises finding the optimal number of devices at processing stages, as well as working volumes or areas of working surfaces of each of these devices. Working volumes and the areas of working surfaces are chosen from a discrete set of standard values. One needs to find an optimal combination of equipment variants using a criterion of optimality, for example, the minimal total capital equipment costs.

The problem of optimal design of multi-product batch plants is a mixed integer nonlinear programming (MINLP) problem [5, 15]. Existing techniques – Monte Carlo method, genetic algorithms, heuristic methods etc. – allow for obtaining suboptimal solutions. Performing an exhaustive search (pure brute-force solution) for finding a global optimum is usually impractical because of the large dimension of the problem. For example, in our earlier work [9], a CES consisting of 16 stages is presented where each process stage can be equipped with devices of 5 to 12 standard sizes. Thus, the number of choices in this case is 5^{16}–12^{16} (which is approximately 10^{11}–10^{17}).

In this paper, we explore the possibility of accelerating the calculations for finding optimal CES designs using a parallelized branch-and-bound algorithm.

Branch-and-bound is a one of the most popular techniques used for solving optimization problems in various fields (e.g., combinatorial optimization, artificial intelligence, etc.). It is also used to solving MINLPs [8]. Branch-and-bound uses a queue of subproblems obtained by decomposing the original problem: it systematically enumerates all solutions and discards a large number of them by using upper and lower bounds of their objective function [3]. In branch-and-bound, the search space is usually considered as a tree, which allows for a structured exploration of the search space. Calculations for the various branches can be carried out simultaneously, which is used to create a parallel version of this method.

Parallel branch-and-bound algorithms have been discussed extensively in the literature. Parallel formulations of depth-first branch-and-bound search are presented in [7]. Martí et al. propose a branch-and-bound algorithm and develop several upper bounds on the objective function values of partial solutions for the Maximum Diversity Problem (MDP) [11]. Mansa et al. analyze the performance of parallel branch-and-bound algorithms with best-first search strategy by examining various anomalies on the expected speed-up [10]. In [6], Gendron et al. present several strategies to exploit parallelism using examples taken from the literature and show that the choice of strategy is greatly influenced by the parallel machine used, as well as by the characteristics of the problem. Rasmussen et al. solve discrete truss topology optimization problems using a parallel implementation of branch-and-bound [16]. In [18], Reinefeld et al. compare work-load balancing strategies of two depth-first searches and propose a scheme that uses fine-grained fixed-sized work packets. Sanders et al. [19] introduce randomized dynamic load balancing algorithms for tree-structured computations, a generalization of backtrack search. Aida [1] et al. discuss the impact of the hierarchical master-worker paradigm on the performance of solving an optimization problem by a parallel branch-and-bound algorithm on a distributed computing system. Bouziane et al. [2] propose a generic approach to embed the master-worker paradigm into software component models and describes how this generic approach can be implemented within an existing software component model. Cauley et al. [4] present a detailed placement strategy designed to exploit distributed computing environments, where the additional computing resources are employed in parallel to improve the optimization time. A Mixed Integer Programming (MIP) model and branch-and-cut optimization strategy are employed to solve the standard cell placement problem. In [21], Zhou at al. present a parallel algorithm for enumerating chemical compounds, which is a fundamental procedure in Chemo- and Bio-informatics.

The problem of optimal design of multi-product batch plants is also covered in the literature. Moreno at al. developed a novel linear generalized disjunctive programming (LGDP) model for the design of multi-product batch plants optimizing both process variables and the structure of the plant through the use of process performance models [13]. Rebennack at al. [17] present a mixed-integer nonlinear programming (MINLP) formulation, where non-convexities are due to the tank investment cost, storage cost, campaign setup cost and variable pro-

duction rates. The objective of the optimization model is to minimize the sum of the production cost per ton per product produced. In [20], Wang at al. present a framework for the design and optimization of multi-product batch processes under uncertainty with environmental considerations.

In this paper, we develop a parallel branch-and-bound algorithm for the globally optimal design of real-world multi-product batch plants, and implement it on modern clusters of multi-core processors.

2 Problem Formulation

A chemical-engineering system (CES) is a set of equipment (reactors, tanks, filters, dryers etc.) which implement the processing stages for manufacturing certain products. Assuming that each processing stage is equipped with a single device, the problem can be formulated as follows:

A CES consists of a sequence of I processing stages. Each processing stage of the system can be equipped with a device from a finite set X_i, with J_i being the number of device variants in X_i. All device variants of a CES are described as $X_i = \{x_{i,j}\}$, $i = \overline{1,I}, j = \overline{1,J_i}$, where $x_{i,j}$ is the main size j (working volume, working surface, etc.) of the device suitable for processing stage i.

Each variant Ω_e, $e = \overline{1,E}$ of a CES, where $E = \prod_{i=1}^{I}(J_i)$ is the number of all possible system variants, is an ordered set of devices work sizes, selected from the respective sets. For example, for a system with 3 processing stages ($I = 3$), the first stage may be equipped with devices selected from a set of 2 working sizes, i. e. $J_1 = 2, X_1 = \{x_{1,1}, x_{1,2}\}$, the second stage from 3 working sizes $J_2 = 3, X_2 = \{x_{2,1}, x_{2,2}, x_{2,3}\}$, and the third stage from 2 working sizes $J_3 = 2, X_3 = \{x_{3,1}, x_{3,2}\}$. Hence, the number of all possible system variants is given by $E = J_1 \cdot J_2 \cdot J_3 = 2 \cdot 3 \cdot 2 = 12$.

As the order of processing stages is predefined, some system variants, e. g., $\{x_{1,1}, x_{2,1}, x_{3,2}\}$, $\{x_{1,2}, x_{2,1}, x_{3,1}\}$ are valid, but others, e. g., $\{x_{3,1}, x_{2,1}, x_{1,2}\}$, $\{x_{2,2}, x_{3,1}, x_{1,1}\}$ are not. Each variant Ω_e of a system should be in operable condition (*compatibility constraint*), i. e. it should satisfy the conditions of a joint action for all its processing stages: $S(\Omega_e) = 0$.

An operable variant of a CES should run at a given production rate in a given period of time (*processing time constraint*), such that it satisfies the restrictions for the duration of its operating period $T(\Omega_e) \leq T_{max}$, where T_{max} is a given maximum period of time.

Thus, designing a multi-product batch plant can be stated as the following optimization problem: to find a variant $\Omega^* \in \Omega_e$, $e = \overline{1,E}$ of a CES, where the optimality criterion – equipment costs $Cost(\Omega_e)$ – reaches a minimum and both compatibility constraint and processing time constraint are satisfied:

$$\Omega^* = argmin\ Cost(\Omega_e), \Omega^* \in (\Omega_e),\ e = \overline{1,E} \tag{1}$$

$$\Omega_e = \{(x_{1,j_1}, x_{2,j_2}, \ldots, x_{I,j_I}) | j_i = \overline{1,J_i}, i = \overline{1,I}\}, e = \overline{1,E} \tag{2}$$

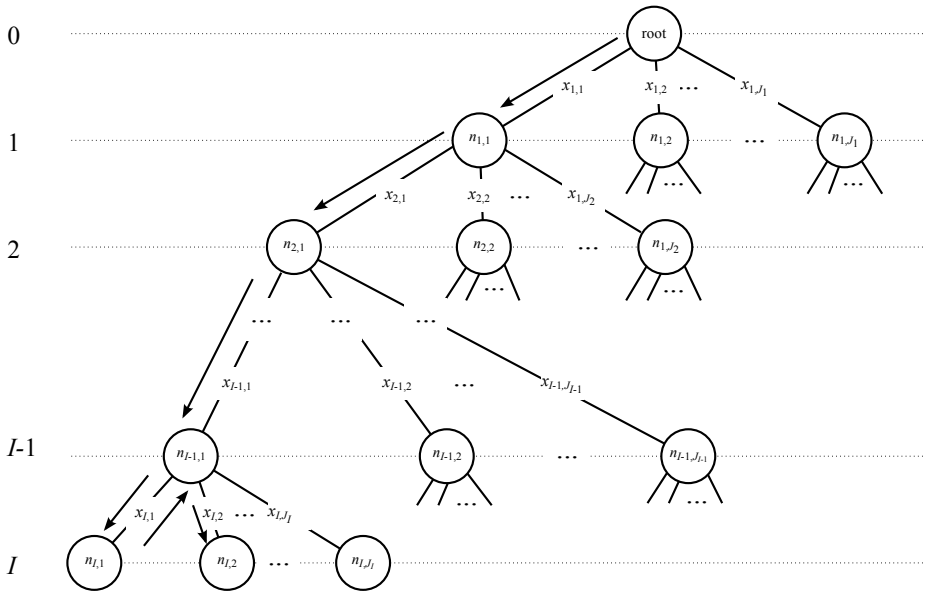


Fig. 1: Tree traversal in depth-first search.

$$x_{i,j} \in X_i,\ i = \overline{1, I}, j = \overline{1, J_i} \tag{3}$$

$$S(\Omega_e) = 0,\ e = \overline{1, E} \tag{4}$$

$$T(\Omega_e) \leq T_{max},\ e = \overline{1, E} \tag{5}$$

In this paper, we use the comprehensive mathematical model of CES operation, including expressions for checking constraints, calculating the optimization criterion, etc., which was initially presented in [9].

3 Sequential Implementation and Its Optimization

In this section, we describe the sequential implementation of a branch-and-bound algorithm for finding an optimal CES.

All possible variants of a CES with I stages can be represented by a tree of height I (see Figure 1). Each level of the tree corresponds to one processing stage of the CES. Each edge corresponds to a selected device variant taken from set X_i, where X_i is the set of possible device variants at stage i of the CES. For example, the edges from level 0 of the tree correspond to elements of X_1. Each node $n_{i,k}$ at the tree layer $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,k}\}$, $i = \overline{1, I}, k = \overline{1, K_i}, K_i = \prod_{l=1}^{i}(J_l)$ corresponds to a variant of a beginning part of the CES, composed of devices

```
1  FindSolution() {  EnumerateVariants(0); }
2
3  /* recursive tree traversal */
4  EnumerateVariants(level) {
5    if (level < I) {
6      for (j = 1; j <= J[level]; j++) {
7        /* append device variant to beginning part */
8        W[level] = X[level, j];
9        /* check compatibility constraint and upper bound */
10       if (S(W) == 0 && PartCost(W, level) < minCost) {
11         /* search recursively */
12         EnumerateVariants (level + 1);  } } }
13   else { /* leaf node */
14     /* check processing time constraint */
15     if (T(W) <= Tmax) {
16       /* check optimality criterion */
17       if (Cost(W) < minCost) {
18         /* make current solution new optimal solution */
19         Wopt = W;
20         minCost = Cost(Wopt); } } }
21 }
```

Listing 1: Sequential implementation of branch-and-bound.

for stages 1 to i of the CES. Each path from the tree's root to one of its leaves thus represents a complete variant of the CES.

To enumerate all possible variants of a CES in the aforementioned tree, a depth-first traversal is performed: starting at level 0 of the tree, all device variants of the CES at a given level are enumerated and appended to the valid beginning parts of the CES. Valid beginning parts are obtained at previous levels, starting with an empty beginning part at level 0. This process continues recursively for all valid beginning parts that result from appending device variants of the current level to the valid beginning parts from previous levels. When a leaf node is reached, the recursive process stops and the current solution is compared to the current optimal solution, possibly replacing it.

Since a complete tree traversal (selecting a device on each edge traversal) and checking constraints (see Equations 4 and 5) would result in considerable computational costs, we use the branch-and-bound technique, with pseudo-code shown in Listing 1. If not stated otherwise, the names of variables correspond to the names in the problem formulation (see Section 2). The tree traversal starts by calling procedure `EnumerateVariants` at level 0 (line 1). This method continues recursively until the optimal CES `Wopt` has been found. Here, `Wopt` is a vector of length `I`, specifying the device variant at each stage of the optimal solution. When traversing the tree, the compatibility constraint (see Equation 4, function `S()`) is checked for the corresponding part of the CES. In addition, we compare the cost for the current beginning part of the CES, consisting of the

first `level` stages (function `PartCost()`) with a global upper bound (variable `minCost`). The initialization of the upper bound is done as sum of all maximum device costs for each productions stage. If the current beginning part of the CES fulfills the compatibility constraint and its costs do not exceed the global upper bound (line 10), we recursively continue tree traversal to the next level (`EnumerateVariants(level + 1)`, line 12). Otherwise we discard deeper levels of the tree and backtrack to the previous level. If a leaf node of the tree is reached (line 13 ff.), the processing time constraint (see Equation 5, function `T()`) is checked for the corresponding CES (line 15). If this constraint is fulfilled, a new solution has been found and its costs (Equation 1, function `Cost()`) are compared to the cost of the last known optimal solution (line 17). If a better solution is obtained, it replaces the previous optimal solution and its costs are taken as new upper bound (line 19–20).

We developed a C++-based implementation of the presented sequential algorithm to perform runtime experiments. As a test case we used the calculation of a CES consisting of 16 processing stages (`I = 16`) with 5 device variants at every stage. Our experiments were conducted on a system comprising 2 Intel Westmere processors (X5650, 6 cores, running at 2.6 GHz) and 4 GB RAM. We use the Intel C++ Compiler version 11.1. We evaluated the execution times of the algorithm's parts to identify the most expensive of them. From the averaged experimental results (Table 1) for our sequential implementation, we observe that the most expensive operation is calling of `T(W)` for checking the processing time constraints of the CES.

Table 1: Averaged execution times of the various algorithm parts.

Algorithm part	Execution time (μs)
Recursive call of `EnumerateVariants()`	0.1
`S(W)`	4.0
`PartCost(W,level)`	0.3
`T(W)`	417.0
`Cost(W)`	0.7

The runtimes presented in the table are quite small for a single computation. But in the searching process with multiple repetitions (billions times) they can add up to tens and hundreds of hours. For our example (16 processing stages with 5 devices variants each), the overall runtime is 27h 11m. In order to reduce the algorithm's runtime, the number of calls of function `T(W)` has to be minimized.

We have implemented the following optimization of the sequential program. From Table 1 we deduce that checking the optimality criterion (`Cost(W)`) is a comparatively cheap operation. If we execute this operation as early as possible, we can discard suboptimal solutions without checking the processing time constraint which is a rather expensive operation. Therefore, we modify the al-

```
12 ...
13 else { /* leaf node */
14   /* check optimality criterion */
15   if (Cost(W) < minCost) {
16     /* check processing time constraint */
17     if (T(W) <= Tmax) {
18       /* make current solution new optimal solution */
19       Wopt = W;
20       minCost = Cost(Wopt); } } }
21 ...
```

Listing 2: Optimizing the sequential algorithm by swapping checks of processing time constraint (slow) and optimality criterion (fast).

gorithm by swapping the checks for the optimality criterion and the processing time constraint (see Listing 2, lines 15, 17)

To evaluate the performance impact of our optimization, we repeat our measurements for the modified implementation using the aforementioned experimental setup. Our simple optimization reduced the runtime approximately by a factor of 2 (13h 52m vs. 27h 11m).

4 Parallel Implementation

The tree-like organization of the branch-and-bound search space provides a potential for the parallelization of our algorithm, as all branches of the tree can be processed simultaneously. In this paper, we use two approaches to parallelize the algorithm: a shared-memory approach and a distributed-memory approach.

4.1 Shared-memory Approach

In the shared-memory approach, all nodes $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,k}\}, i = \overline{1, I}, k = \overline{1, K_i}, K_i = \prod_{l=1}^{i} J_l$ at each layer i of the tree are regarded as independent *tasks* that can be executed in parallel. The total number of tasks, $N_{tasks} = \sum_{i=1}^{G} K_i$, can be a very large number. Therefore, a *granularity* parameter G is introduced to limit the degree of parallelism to a certain level of the tree: subtrees below the granularity level are not split into tasks but rather processed sequentially.

A pseudo-code for this approach is given in Listing 3. The main difference as compared to the sequential version is that recursive function calls are performed by newly created concurrent tasks (line 12). Besides, a copy of the current beginning part of the CES, `W`, has to be provided to each task.

The merit of this approach is its simple implementation: no communication is needed between tasks as they rely on shared memory for data exchange.

```
1  FindSolution() { EnumerateVariants(0); }
2
3  EnumerateVariants(level) {
4    if (level < I) {
5      for (j=1; j <= J[level]; j++) {
6        /* append device variant to beginning part */
7        W[level] = X[level, j];
8        /* check compatibility constraint and upper bound */
9        if (S(W) == 0 && PartCost(W, level) < minCost) {
10         if (level < G) { /* check granularity */
11           /* create concurrent task */
12           CREATE TASK: EnumerateVariants(level + 1);  }
13         else {
14           /* search recursively */
15           EnumerateVariants(level + 1); } } } }
16   else { /* leaf node */
17     ... }
18 }
```

Listing 3: The shared-memory approach for parallel branch-and-bound.

4.2 Distributed-memory Approach

We use the master-worker paradigm for an alternative, distributed-memory parallelization of our algorithm: a single *master* process dispatches a subset of computations to multiple *worker* processes and gathers computed results from them.

Master Process The master (see Listing 4) performs a depth-first traversal of the tree using a recursive procedure `MasterEnumerateVariants` to some level G (*granularity*), $1 \leq G \leq I$. Using this procedure, the master creates beginning parts $W[i], i = \overline{1, G}$ of the CES (lines 24–29). At the last level of recursion, the master waits for worker messages (line 32), which can be of two types: *solution* (SOLUTION) or *job request* (REQUEST_WORK). If the master receives a *solution* message (line 33), the costs of the received solution are compared to the costs of the current optimal solution (optimality criterion, line 35). If a better solution has been found by the worker, it is stored and replaces the current optimal solution (lines 36–38). When a job request is received (line 39), the master responds by sending *job message* (DO_WORK) containing the current beginning part of the CES and the current optimal solution to the worker (line 40). Afterwards, a new beginning part of the CES is generated to be passed to a worker (lines 25–29). If no new beginning part of the CES can be generated, the master returns from the recursive procedure `MasterEnumerateVariants` (line 6). The master continues receiving solutions from workers and compares them to the optimal solution. However, if a worker sends a job request, the master sends a *quit* message (QUIT) to the worker, to terminate the worker process. After quit messages have been sent to all workers, the master process ends.

```
1  Master() {
2    /* number of workers (one of processes is master) */
3    num_workers = NUM_PROCESSORS - 1;
4
5    /* start tree traversal */
6    MasterEnumerateVariants(0);
7
8    /* wait for remaining solutions and stop workers */
9    while (num_workers > 0) {
10     msg = ReceiveWorkerMessage();
11     if     (msg.type == SOLUTION) {
12       /* check optimality criterion */
13       if (Cost(msg.W) < minCost) {
14         /* make solution new optimal solution */
15         Wopt = msg.W;
16         minCost = Cost(msg.W);  } }
17     elseif (msg.type == REQUEST_WORK) {
18       /* stop worker */
19       SendWorkerMessage(msg.workerID, QUIT);
20       num_workers--;  } }
21 }
22
23 MasterEnumerateVariants(level) {
24   if (level < G) { /* check granularity */
25     for (j=1; j <= J[level]; j++) {
26       W[level] = X[level, j];
27       if (S(W) == 0 && PartCost(W,level) < minCost) {
28         /* search recursively */
29         MasterEnumerateVariants(level + 1); } } }
30   else {
31     while (true) {
32       msg = ReceiveWorkerMessage();
33       if     (msg.type == SOLUTION) {
34         /* check optimality criterion */
35         if (Cost(msg.W) < minCost) {
36           /* make solution new optimal solution */
37           Wopt = msg.W;
38           minCost = Cost(msg.W);  } }
39       elseif (msg.type == REQUEST_WORK) {
40         SendWorkerMessage(msg.workerID, W, minCost);
41         break;  } } }
42 }
```

Listing 4: Distributed-memory approach: Pseudo-code of master.

Worker Process The worker (see Listing 5) starts by sending a job request to the master (line 3) and waits for the response. The response can be of one of two types: *job message* (DO_WORK) or *quit* (QUIT). If a job message comprising a beginning part of the CES and the current upper bound of the optimality criterion is received, the worker calls the recursive procedure `WorkerEnumerateVariants` (line 5–8). Within this procedure, the worker traverses the remaining sub-tree $W[i], i = \overline{G+1, I}$ of the received CES' beginning part to find solutions in the same way the sequential algorithm does (lines 5–20 of Listing 1). If the worker finds a solution which costs do not exceed the upper bound of the optimality criterion (lines 24–27), it makes this solution the new optimal solution (lines 28–31). When the recursive procedure ends, the worker sends its new optimal solution, if any, to the master (line 9–11) and requests a new job. If a quit message is received, the worker process terminates (line 8–9).

The distributed-memory approach is more difficult to implement than the shared-memory approach: master-worker communication has to be specified explicitly in order to exchange data in a distributed-memory system. Besides, a single master constitutes a possible performance bottleneck of this implementation.

5 Experimental Results

To study the speedup of our two parallelization approaches, we created two corresponding implementations and conducted runtime experiments on a heterogeneous cluster consisting of:

- 36 nodes with 2 quad-core processors (Intel X5550 Nehalem, running at 2.6 GHz) with 3 GB RAM each,
- 198 nodes with 2 hexa-core processors (Intel Westmere X5650, running at 2.6 GHz) with 2 or 4 GB RAM each, and
- 4 nodes with 4 eight-core processors (Intel Xeon E7550, running at 2 GHz) with 128 GB RAM each.

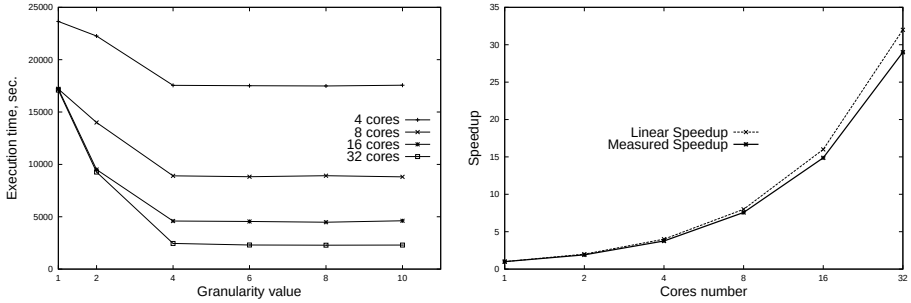
The nodes are interconnected via Infiniband. Programs were compiled using the Intel C++ Compiler 11.1.

We study the design of a CES consisting of 16 processing stages with 5 variants of devices at every stage as test case. The implementations are written in C++ using OpenMP version 3.0 (Open Multi-Processing) [14] for the task-based approach (see Section 4.1), and the Message Passing Interface (MPI) [12] for the master-worker approach (see Section 4.2).

The implementation using OpenMP is derived from the sequential implementation by inserting directives: a `parallel` construct with a nested `single` construct is put around the call of the `EnumerateVariants` (line 1 of Listing 3), such that one of these threads starts the recursive tree traversal, while the other threads stay idle. Within the recursive procedure new tasks are created using

```
1  Worker() {
2    while (true) {
3      SendMasterMessage(workerID, REQUEST_WORK);
4      msg = ReceiveMasterMessage();
5      if (msg.type == DO_WORK) {
6        minCost = msg.minCost;
7        foundNewSolution = false;
8        WorkerEnumerateVariants(G + 1);
9        if (foundNewSolution) {
10         /* send new optimal solution to master */
11         SendMasterMessage(workerID, SOLUTION, Wopt);  } }
12     elseif (msg.type == QUIT) {
13       break;  } }
14 }
15
16 WorkerEnumerateVariants(level) {
17   if (level < I) {
18     for (j=1; j <= J[level]; j++) {
19       /* append device variant to beginning part */
20       W[level] = X[level, j];
21       /* check compatibility constraint and upper bound */
22       if (S(W) == 0 && PartCost(W, level) < minCost) {
23         /* search recursively */
24         WorkerEnumerateVariants(level + 1); } } }
25   else { /* leaf node */
26     /* check optimality criterion */
27     if (Cost(W) < minCost)) {
28       /* check processing time constraint */
29       if (T(W) <= Tmax) {
30         /* make solution new (local) optimal solution */
31         Wopt = W;
32         minCost = Cost(Wopt);
33         foundNewSolution = true;  } } }
34 }
```

Listing 5: Distributed-memory approach: Pseudo-code of worker.



(a) Runtime depending on granularity. (b) Speedup.

Fig. 2: Experimental results for the OpenMP-based implementation.

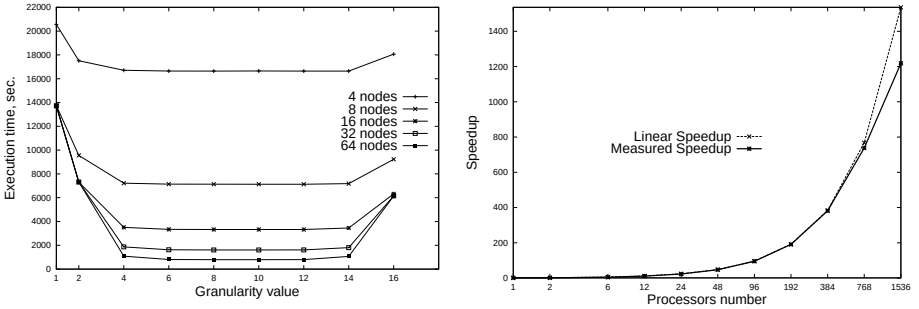
the `task` construct of OpenMP. While the first thread continues creating tasks, the other threads process these tasks.

We run our OpenMP-based implementation on a single node consisting of 4 CPUs with altogether 32 cores, setting granularity values from 1 to 10. We observed that for granularity greater than 10, too many tasks were created, such that the implementation ran out of memory. The results are shown in Figure 2a. Figure 2b shows the speedup of our OpenMP-based implementation using up to 32 cores. Granularity has been set to 10.

In our master-worker implementation, we use MPI's point-to-point communication functions `send` and `recv` for exchanging messages between master and worker. We performed the same measurements on up to 64 Westmere nodes. Here, we also observed best performance for granularity values from 4 to 14 (see Figure 3a). The minimum number of processors for running the program is two (master and one worker). While there is no speedup when using 2 processors, it increases nearly linearly when using up to 768 processors. With greater numbers of processors, the growth of speedup slows down. The performance of the master process may become a bottleneck of application performance when it controls too many worker processes, because the master frequently communicates with all workers.

Both implementations provide high scalability. On the same hardware, the performance of both approaches differs slightly. However, in spite of its more difficult implementation, the MPI implementation is preferable, because it runs both on shared-memory machines and on computers with distributed memory. Currently, shared-memory machines with more processors (hundreds and thousands) are rare, unlike computing clusters. Also with a large number of tasks we may not have enough memory as in our case when $G > 10$.

Selecting a suitable granularity value is crucial for optimal performance. Usually, granularity should be set to a value, such that the number of initial parts for a system is significantly greater than the number of processors, i. e. $\prod_{i=0}^{G} J_i \gg N_p$. However, the distribution of initial parts to processors may be-



(a) Runtime depending on granularity. (b) Speedup.

Fig. 3: Experimental results for the MPI-based implementation.

come unbalanced if initial parts for a systems are discarded early by the branch-and-bound paradigm. Hence, we empirically determined a factor to optimize load balance. For the above example (16 processing stages with 5 device variants at each), this factor is 2–3, such that a sensible granularity value G is within $2 \cdot \log_5 N_p \leq G \leq 3 \cdot \log_5 N_p$.

6 Conclusion

We proposed two approaches to implement a parallel branch-and-bound algorithm for solving the optimization problem for multi-product batch plants. Implementations of our approaches based on OpenMP and MPI have been presented. Runtime experiments for our implementations using a real-world example of a multi-product batch plant show that our solution provides considerable speedup. This is well correlated with experimental results obtained in, e. g., [6], where also near-linear speedups were observed. Both implementations provide good parallel scalability.

We also analyzed the impact of the degree of parallelism controlled by a granularity parameter. From our results we conclude that while the MPI-based implementation suffers a communication bottleneck for large numbers of processors (the reasons for that and methods of overcoming are described in detail in [1]), it still provides better performance and flexibility as compared to the OpenMP-based implementation.

In future work we will investigate the use of a hierarchical master-worker implementation, in order to reduce the communication bottleneck which we observed in our current implementation. This paper presents a parallel version only of the branch-and-bound algorithm. In addition, quite interesting would be the parallelization of comprehensive mathematical model of CES operation. This problem requires also deeper and more detailed research in further works.

Acknowledgement

This work was supported by the DAAD (German Academic Exchange Service) and by the Ministry of Education and Science of the Russian Federation under "Mikhail Lomonosov II"-Programme.

References

1. Aida, K., Natsume, W., Futakata, Y.: Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03) pp. 156–164 (2003)
2. Bouziane, H.L., Pérez, C., Priol, T.: Extending software component models with the master-worker paradigm. Parallel Computing 36(2-3), 86–103 (February 2010)
3. Brassard, G., Bratley, P.: Fundamentals of Algorithmics. Prentice Hall (1996)
4. Cauley, S., Balakrishnan, V., Hu, Y.C., Koh, C.K.: A parallel branch-and-cut approach for detailed placement. ACM Transactions on Design Automation of Electronic Systems (TODAES) 16(2), 18:1–18:19 (April 2011)
5. El Hamzaoui, Y., Hernandez, J., Cruz-Chavez, M., Bassam, A.: Search for Optimal Design of Multiproduct Batch Plants under Uncertain Demand using Gaussian Process Modeling Solved by Heuristics Methods. Berkeley Electronic Press (2010)
6. Gendron, B., Crainic, T.G.: Parallel branch-and-bound algorithms: Survey and synthesis. Operations Research 42(6), 1042–1066 (1994)
7. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing, Design and Analysis of Algorithms. Addison-Wesley, second edn. (2003)
8. Leyffer, S., Linderoth, J., Luedtke, J., Miller, A., Munson, T.: Applications and algorithms for mixed integer nonlinear programming. Journal of Physics: Conference Series 180(1), 12–14 (2009)
9. Malygin, E., Karpushkin, S., Borisenko, A.: A mathematical model of the functioning of multiproduct chemical engineering systems. Theoretical foundations of chemical engineering 39(4), 429–439 (2005)
10. Mansa, B., Roucairol, C.: Performances of parallel branch and bound algorithms with best-first search. Discrete Applied Mathematics 66(1), 57–74 (1996)
11. Martí, R., Gallego, M., Duarte, A.: A branch and bound algorithm for the maximum diversity problem. European Journal of Operational Research 200(1), 36–44 (2010)
12. Message Passing Interface Forum: Message Passing Interface Standards Documents, http://www.mpi-forum.org
13. Moreno, M.S., Montagna, J.M.: Multiproduct batch plants design using linear process performance models. American Institute of Chemical Engineer Journal 57(1), 122–135 (2011)
14. OpenMP Architecture Review Board: The OpenMP API specification for parallel programming, http://www.openmp.org
15. Ponsich, A., Azzaro-Pantel, C., Domenech, S., Pibouleau, L.: Mixed-integer nonlinear programming optimization strategies for batch plant design problems. Industrial & Engineering Chemistry Research 46(3), 854–863 (2007)
16. Rasmussen, M., Stolpe, M.: Global optimization of discrete truss topology design problems using a parallel cut-and-branch method. Computers & Structures 86(13-14), 1527–1538 (2008)

17. Rebennack, S., Kallrath, J., Pardalos, P.M.: Optimal storage design for a multi-product plant: A non-convex minlp formulation. Computers & Chemical Engineering 35(2), 255 – 271 (2011)
18. Reinefeld, A., Schnecke, V.: Work-load balancing in highly parallel depth-first search. Scalable High-Performance Computing Conference pp. 773–780 (1994)
19. Sanders, P.: Better algorithms for parallel backtracking. Parallel Algorithms for Irregularly Structured Problems. Lecture Notes in Computer Science 980, 333–347 (1995)
20. Wang, Z., Jia, X.P., Shi, L.: Optimization of multi-product batch plant design under uncertainty with environmental considerations. Clean Technologies and Environmental Policy 12, 273–282 (2010)
21. Zhou, J., Yu, K.M., Lin, C., Shih, K.C., Tang, C.: Balanced multi-process parallel algorithm for chemical compound inference with given path frequencies. In: Hsu, C.H., Yang, L., Park, J., Yeo, S.S. (eds.) Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science, vol. 6082, pp. 178–187. Springer Berlin / Heidelberg (2010)