Supplementary File - A Novel Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms

Hamidreza Khaleghzadeh, Ravi Reddy, and Alexey Lastovetsky, *Member, IEEE*

This supplementary file contains the supporting materials of the TPDS manuscript, "A Novel Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms". They are:

- Experimental methodology followed to obtain speed functions.
- Use case containing matrix-vector multiplication on a homogeneous cluster of Intel Xeon Phi co-processors.
- Load balancing and load imbalancing algorithms.
- Comparison of actual and simulated execution times.
- Using *HPOPTA* for data partitioning on simulated clusters of heterogeneous nodes.
- Description of the helper functions used in the algorithm, *HPOPTA*.
- Correctness and complexity proofs of *HPOPTA*.

1 Experimental Methodology to Build the Performance Functions

We followed the methodology described below to make sure the experimental results are reliable:

- The server is fully reserved and dedicated to these experiments during their execution. We also made certain that there are no drastic fluctuations in the load due to abnormal events in the server by monitoring its load continuously for a week using the tool *sar*. Insignificant variation in the load was observed during this monitoring period suggesting normal and clean behaviour of the server.
- Our heterogeneous application is executed simultaneously on all the three abstract processors, CPU, GPU and Xeon Phi. To obtain a data point in the speed functions, the application is repeatedly executed until the sample means of execution times for all the abstract processors lie in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by plotting the distributions of observations.

- *H. Khaleghzadeh, R. Reddy and A. Lastovetsky are with the School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland. E-mail: hamidreza.khaleghzadeh@ucdconnect.ie, ravi.manumachu@ucd.ie, alexey.lastovetsky@ucd.ie*

The function $MeanUsingTtest$, shown in Algorithm 1, describes the execution of an application app to satisfy the statistical confidence. For each data point in the speed functions, the function is invoked to repeatedly execute the application app until one of the following three conditions is satisfied:

1) The maximum number of repetitions ($maxReps$) have been exceeded (Line 3).
2) The sample means of all devices fall in the confidence interval (or the precision of measurement eps has been achieved) (Lines 9-12).
3) The elapsed time of the repetitions of application execution has exceeded the maximum time allowed ($maxT$ in seconds) (Lines 13-15).

So, for each data point, the function $MeanUsingTtest$ is invoked and the sample means t_{cpu}, t_{gpu}, and t_{phi} are returned at the end of invocation. The input minimum and maximum number of repetitions, $minReps$ and $maxReps$, differ based on the problem size solved. For small problem sizes ($32 \leq n \leq 1024$), these values are set to 10000 and 100000 respectively. For medium problem sizes ($1024 < n \leq 5120$), these values are set to 100 and 1000. For large problem sizes ($n > 5120$), these values are set to 5 and 50. The values of $maxT$, cl, and eps are respectively set to 3600, 0.95, and 0.025. The heterogeneous application, app, returns the times elapsed by computational kernels on CPU, GPU and Xeon Phi whenever finishes its execution (Line 4). The execution times are stored in arrays $exec_{cpu}$, $exec_{gpu}$ and $exec_{phi}$.

The helper function, $CalAccuracy$, Algorithm 2, returns 1 if the sample mean of execution times (in $Array$) lies in the 95% confidence interval (cl) and a precision of 0.025 ($eps = 2.5\%$) has been achieved. Otherwise, it returns 0.

If the precision of measurement is not achieved before the maximum number of repeats have been completed, we increase the number of repetitions and also the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in our experiments.

Algorithm 1 Function Determining the Mean of an Experimental Run using Student's t-test.

procedure MEANUSINGTTEST($app, minReps, maxReps, maxT, cl, accuracy, repsOut, clOut, timeOut, accuracyOut, mean$)
Input:
The application to execute, app
The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$
The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$
The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$
The required confidence level, $cl \in \mathbb{R}_{>0}$
The required accuracy, $eps \in \mathbb{R}_{>0}$
Output:
The number of experimental runs actually made, $repsOut \in \mathbb{Z}_{>0}$
The elapsed time, $etimeOut \in \mathbb{R}_{>0}$
The mean execution times, $t_{cpu}, t_{gpu}, t_{phi} \in \mathbb{R}_{>0}$

$reps \leftarrow 0$; $stop \leftarrow 0$; $sum \leftarrow 0$; $etime \leftarrow 0$
while ($reps < maxReps$) and (!$stop$) **do**
  $(exec_{cpu}[reps], exec_{gpu}[reps], exec_{phi}[reps]) \leftarrow$ EXECUTE(app)
  $sum_{cpu} += exec_{cpu}[reps]$
  $sum_{gpu} += exec_{gpu}[reps]$
  $sum_{phi} += exec_{phi}[reps]$
  if $reps > minReps$ **then**
    $stop_{cpu} \leftarrow$ CALACCURACY($cl, reps + 1, exec_{cpu}, eps$)
    $stop_{gpu} \leftarrow$ CALACCURACY($cl, reps + 1, exec_{gpu}, eps$)
    $stop_{phi} \leftarrow$ CALACCURACY($cl, reps + 1, exec_{phi}, eps$)
    $stop \leftarrow stop_{cpu} \wedge stop_{gpu} \wedge stop_{phi}$
    if $\max\{sum_{cpu}, sum_{gpu}, sum_{phi}\} > maxT$ **then**
      $stop \leftarrow 1$
    end if
  end if
  $reps \leftarrow reps + 1$
end while
$repsOut \leftarrow reps$;
$etimeOut \leftarrow \max\{sum_{cpu}, sum_{gpu}, sum_{phi}\}$
$t_{cpu} \leftarrow \frac{sum_{cpu}}{reps}$; $t_{gpu} \leftarrow \frac{sum_{gpu}}{reps}$; $t_{phi} \leftarrow \frac{sum_{phi}}{reps}$
end procedure

Algorithm 2 Algorithm calculating accuracy

function CALACCURACY($cl, reps, Array, eps$)
  $clOut \leftarrow$ fabs(gsl_cdf_tdist_Pinv($cl, reps - 1$))
    $\times$ gsl_stats_sd($ObjArray, 1, reps$)
    / sqrt($reps$)
  if $clOut \times \frac{reps}{sum} < eps$ **then**
    return 1
  end if
  return 0
end function

2 MATRIX-VECTOR MULTIPLICATION ON HOMOGENEOUS CLUSTER OF INTEL XEON PHI PROCESSORS

In this section, we consider the execution of a matrix-vector multiplication application executing the highly optimized multi-threaded Intel MKL DGEMV routine in a homogeneous cluster of six nodes where each node contains two Intel Xeon Phi accelerators. So, altogether, there are twelve identical Xeon Phi co-processors. The specification of the accelerator is shown in the Table 1. The application multiplies a dense matrix of size $N \times N$ with a vector of size N. Figure 1 shows the performance profiles of the application for all the twelve accelerators. The application runs on all cores of each Intel Xeon Phi co-processor. These profiles were built simultaneously to take into account resource contention.

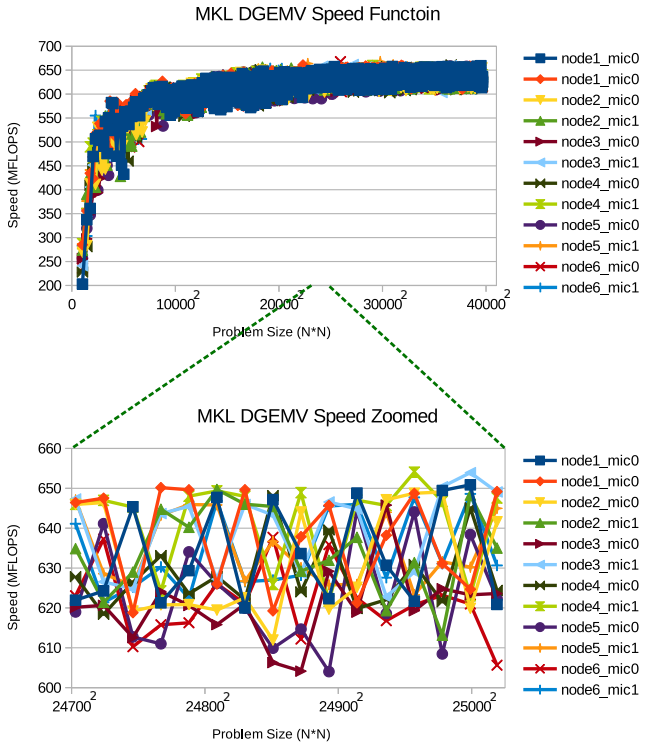

Fig. 1. Speeds of Intel MKL DGEMV application for twelve Intel Xeon Phi SE10/7120 series coprocessors.

TABLE 1
Specification of the Intel Xeon Phi coprocessor SE10/7120 series.

Technical Specifications	Intel Xeon Phi SE10/7120 series
No. of processor cores	61
Base frequency	1333 MHz
Total main memory	15 GB GDDR5
L2 cache size	30.5 MB
Memory bandwidth	352 GB/sec
Memory clock	2750000 kHz
TDP	300 W
Idle Power	98 W

The figure also shows the zoomed speed function between two arbitrarily chosen points in the speed functions.

To make sure that the experimental results are reliable, we follow the experimental methodology described earlier. It contains the following main steps: 1) We make sure the platform is fully reserved and dedicated to our experiments and is exhibiting clean and normal behaviour by monitoring its load continuously for a week. 2) For each data point in the speed functions of an application, the sample mean is used, which is calculated by executing the application repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by checking the density plots of the observations.

From the figure, we can observe the following:

- It is evident that the shapes violate the assumptions

on shape of FPMs. Therefore, load balancing data partitioning algorithms based on FPMs may not return optimal solutions.

- Although all the Xeon Phi co-processors are identical, their speed functions demonstrate different shapes with noteworthy variations. The average of the variations is 7%. Therefore, due to the inherent nature of the platforms today, even in a homogeneous environment, one must use heterogeneous workload distribution for optimal performance. In this case, using an average discrete speed function as input to data partitioning algorithm as done in [1] may not be optimal. Therefore, the new model-based methods proposed in [1], [2] cannot be used since they consider all identical processors and take a single speed function as an input.

3 Load Balancing and Load Imbalancing Algorithms

In this section, we explain what we mean by load balancing algorithms and load imbalancing algorithms. *HPOPTA* is a load imbalancing algorithm. Having said this, if the optimal workload distribution load balances the application, then *HPOPTA* finds it.

We define a load balancing algorithm as one that determines the workload distribution where the problem sizes allocated to the processors are proportional to their speeds. The intuition behind load balancing is that balancing the application improves its performance in the following manner: a balanced application does not waste processor cycles on waiting at points of synchronization and data exchange, maximizing this way the utilization of the processors and minimizing the computation time.

Lastovetsky et al. [1] present a formal study of load balancing. In this work, they show that in order to guarantee that the balanced configuration of the application will execute the workload of size n faster than any unbalanced configuration, the speed functions $s_i(x)$, characterizing the performance profiles of the processors, should satisfy the condition:

$$\forall \Delta x > 0 : \frac{s_i(x)}{x} \geq \frac{s_i(x + \Delta x)}{x + \Delta x}$$

The speed $s_i(x)$ is calculated as $\frac{x}{t_i(x)}$, where $t_i(x)$ is the execution time of the workload of size x on processor i. This condition means that the increase of the workload, x, will never result in the decrease of the execution time.

However, in this work, we show that this condition is violated by the performance profiles of the data-parallel applications executing on modern HPC platforms.

HPOPTA is designed to deal with the shapes of performance profiles where the condition is no longer satisfied. We call such an algorithm, *load imbalancing algorithm*, where it determines the optimal workload distribution that minimizes the execution time of computations of a data-parallel application but which does not load balance the application.

We illustrate using an example. Consider a platform consisting of four abstract processors ($p = 4$) with speed functions presented in Section 4. Let the workload size to be solved be equal to 31 ($n = 31$). In this example, load balanced solution is $\{(2, 13), (11, 13), (9, 13), (9, 13)\}$ and the load balanced execution time therefore is 13. However, the optimal solution found by *HPOPTA* is $\{(9, 3), (9, 3), (7, 1), (6, 2)\}$ with the optimal execution time of 3. It is obvious that the optimal solution does not balance the load between processors.

4 Comparison of Actual and Simulated Execution Times

In the introduction section, when we present the modelling of the three abstract processors in our applications, we mention that while performance models are built where the data points for the same problem size are obtained simultaneously, during the actual execution of the data-parallel application using the workload distribution determined by our data partitioning algorithm, the problem sizes executed by the abstract processors can be different. This is because different processors can be allocated different problem sizes by our heterogeneous data partitioning algorithm. Later in Experimental analysis of *HPOPTA*, we evaluated our algorithm and extracted the execution times from the performance profiles.

In this section, we experimentally show that the execution times of these problem sizes simultaneously would not differ significantly from those present in the performance models. We experiment with Matrix Multiplication and 2D FFT, configured for execution on *HCLServer* as explained in the introduction section. We compare the execution times of solutions returned by *HPOPTA* with actual execution times on *HCLServer*. To obtain the actual results, a parallel application is executed where each processor is allocated the problem size given by *HPOPTA*, and its parallel execution time is measured. To obtain the actual experimental results, we have followed the experimental methodology explained in this supplemental. Since *HPOPTA* considers all the possible combinations of workload distributions, even combinations where one or more processors are allocated problem size of zero, the number of processors used in the experiment ranges from 1 to 3.

We create an experimental data set for Matrix Multiplication whose data points ranges from 64^2 to 80000^2 with step size of 64^2. Figure 2 shows the execution time of Matrix Multiplication on *HCLServer* when executed using *HPOPTA* ($HPOPTA\ Actual\ Time$) with the simulated execution time ($HPOPTA\ Simulation\ Time$).

To analyse FFT, the experimental data set includes data points ranging from 16^2 to 64000^2 with step size of 16^2. Figure 3 compares actual with simulated execution times for FFT.

From the figures, one can see that the differences between simulated results and actual execution time are insignificant.

5 Using *HPOPTA* for Data partitioning on Simulated Clusters of Heterogeneous Nodes

In this section, to study the performance improvements given by *HPOPTA* at scale, we simulated clusters consisting of $8, 16, \cdots, 256$ *HCLServer* nodes, where each node has

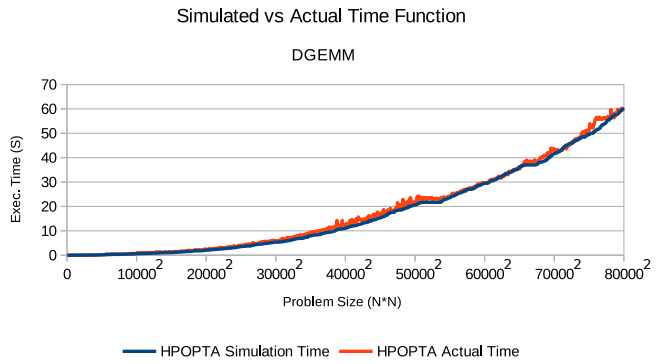

Fig. 2. Comparison of actual with simulated execution times for Matrix Multiplication on *HCLServer*.

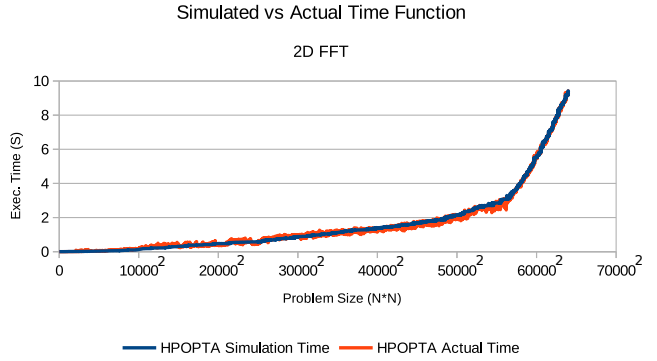

Fig. 3. Comparison of actual with simulated execution times for 2D FFT on *HCLServer*.

three abstract processors and therefore the total number of heterogeneous abstract processors range from 24 to 768. We conduct experiments that are a combination of actual measurements conducted on *HCLServer* and simulations for clusters containing replicas of *HCLServer*. The actual measurements include the construction of time functions, which are input to *HPOPTA* (refer section "Data Partitioning on Hybrid Server" in the manuscript). The simulations contain the execution of *HPOPTA* to determine workload distributions, which allow us to calculate the parallel execution times of computations in the data-parallel applications and consequently the speedups demonstrated by *HPOPTA*.

HPOPTA requires as input, the time function of each abstract processor in the simulated cluster. Since all nodes are identical, we build the time functions for one node, *HCLServer*, and then use them for all nodes in the simulated cluster. For example, for a simulated cluster consisting of 8 *HCLServer* nodes, the input to *HPOPTA* will consist of 24 (3×8) time functions.

We now examine *HPOPTA* on simulated clusters of heterogeneous nodes using the same data parallel applications, Matrix Multiplication and FFT.

5.1 Matrix Multiplication

For each simulated cluster, we execute DGEMM using a test data set whose data points ranges from $(\frac{p}{3} \times 64 \times 100)^2$ to $(p \times 64 \times 700)^2$ with step size of 64^2. The obtained results show that *HPOPTA* gives the minimum, average, and maximum percentage improvements of 0, 14, and 261 percent respectively in comparison with *FPM*.

We choose the same problem sizes 4736, 28672, and 44800 to obtain relative speeds for *CPM* workload distribution. The average percentage improvements of *HPOPTA* over *CPM* are 122, 106, and 82 percent respectively.

5.2 FFT

To analyse FFT, the experimental data set include data points ranging from $(\frac{p}{3} \times 16 \times 100)^2$ to $(p \times 16 \times 1500)^2$ with step size of 16^2. The obtained results show *HPOPTA* gives the minimum, average and maximum percentage of improvement of 0, 43, 513 percent respectively in comparison with *FPM*.

We choose the same problem sizes 4320, 13824, and 24000 to obtain relative speeds for *CPM* workload distribution. The average percentage of improvement of *HPOPTA* over *CPM* are 301, 164 and 129 percent respectively.

5.3 Discussion

We observed almost the same percentage of improvement for different cluster sizes for both DGEMM and FFT. It can be concluded that the performance improvement is independent of p assuming the cost of communications is not taken into account.

There is a strong correlation between average performance improvements and the average variations in speed functions. Furthermore, the maximum performance improvement cannot exceed the maximum variation in the speed functions. In our experiments, all nodes in simulated clusters are identical and their speed functions consequently will be identical. Thus, average and maximum performance improvements of a simulated cluster consisting of identical nodes are not related to the number of nodes but related to the shapes of speed functions which are identical for all nodes.

We would like to mention that the study and incorporation of cost of communications is a significant body of work and is therefore out of scope of this paper. It is the focus of our current research.

In addition, the number of abstract processors in the optimal solution determined by *HPOPTA* is often less than p. For example, in a cluster of eight *HCLServer* nodes, the optimal solution for FFT for matrix size 304×304 uses just one GPU while the other 23 abstract processors are given zero problem size. For FFT for matrix size 25552×25552 the optimal workload distribution uses 21 abstract processors leaving one CPU and two Xeon Phis unused.

5.4 Hierarchical Two-level Workload Distribution

In Section 5, we used *HPOPTA* for optimal workload distribution in a cluster of identical hybrid nodes. As *HPOPTA* is oblivious of the regular structure of the underlying platform, in order to find an optimal solution for a cluster of h nodes it had to analyse $3 \times h$ time functions. In this section, we present an hierarchical workload distribution algorithm, *HiPOPTA*, which combines *HPOPTA* and *POPTA* [2] to find an optimal solution for a cluster of h identical nodes only

using $c+1$ time functions instead of $c \times h$, where c is the number of heterogeneous processors in one node.

HiPOPTA first distributes workload between identical nodes (Inter-node workload distribution) using *POPTA*. The input to it is a whole speed function of a node constructed using *HPOPTA*. The assigned problem size to each node is then distributed between the processing elements of each node (Intra-node workload distribution) using *HPOPTA*.

We explain the steps of *HiPOPTA* using a cluster of *HCLServer* nodes:

- **Building speed function of whole *HCLServer* using *HPOPTA*:** For each workload size, we run the heterogeneous application on *HCLServer* using the *HPOPTA* workload distribution and measure its parallel execution time. The resulting speed function characterizes the performance of *HCLServer* as a whole. Since all the nodes in the simulated cluster are identical, their speed functions will be the same, too. Figures 14 and 16 respectively show the speed functions of Matrix Multiplication and 2D FFT of whole *HCLServer*. In Section "Data Partitioning on Hybrid Server" in the manuscript, we have explained in detail how these speed functions are built for *HCLServer*.
- **Inter-node workload distribution:** We use the whole *HCLServer* speed function to distribute workload between the nodes of the simulated cluster. Since all nodes are identical, we can use *POPTA* [2], an algorithm to find optimal workload distribution on *homogeneous* platforms, for finding the optimal workload distribution between nodes.
- **Intra-node workload distribution:** *HPOPTA* is then applied inside each node to divide the assigned workload between CPU, GPU, and Xeon Phi of this node so that the execution time remains minimum. The intra-node workload distributions can be determined by running *HPOPTA* on the nodes of the cluster in parallel.

To evaluate *HiPOPTA*, we repeat experiments from Section 5. We use Matrix Multiplication and FFT with the same experimental data sets as explained in Section 5. As expected, the resulting execution times of the distributions returned by *HiPOPTA* are the same as the ones obtained in the section 5 using plain *HPOPTA*.

The reason behind the use of two-level partitioning is the reduction in theoretical and practical complexity for finding optimal distributions on large scale clusters. Assume a cluster involving h identical nodes where each node consists of c processors. Therefore, the cluster totally comprises $c \times h$ processors ($p = c \times h$). Let cardinality of time functions be m where $c >> m$ and $h >> m$. We first calculate the time complexity of *HiPOPTA*. There are h identical nodes and therefore *POPTA* finds the optimal inter-node distribution with the time complexity of $O(h^2)$ [2]. Optimal intra-node workload distributions are then found using parallel executions of *HPOPTA* on h nodes with time complexity of $O(c^3)$. Therefore, the total theoretical complexity will be equal to $O(h^2 + c^3)$. The theoretical complexity of the non-hierarchical partitioning is $O(p^3)$, which is equal to $O(c^3 \times h^3)$. Therefore, *HiPOPTA* is $O(\frac{c^3 \times h^3}{h^2 + c^3})$ times faster than the non-hierarchical one. In addition, the hierarchical workload distribution allows parallel computations to find optimal distribution.

HiPOPTA always returns an optimal distribution. Indeed, according to [2], *POPTA* finds an optimal workload distribution between identical compute nodes represented by their speed function. Assuming that the speed function of a node reflects the fastest speed of execution of any given workload, it will find a globally optimal distribution. However, by construction, the speed function of a node as a whole found locally by *HPOPTA* does give the fastest possible speed of execution for any workload given to the node. Note that since there may be more than one optimal distribution, distributions returned by *HiPOPTA* and *HPOPTA* may be different. However, their execution times will be always the same.

6 Helper Routines Called in *HPOPTA*

6.1 Function *GetTime*

The function GETTIME(T_i, w) returns the execution time of workload size w from the time function T_i (Algorithm 3). It returns 0 when input workload size w is 0. If there is no match for w in the time function T_i, the function returns -1.

Algorithm 3 Algorithm Finding the execution Time of a Given Problem Size

```
1: function GETTIME(T_i, w)
2:     if w = 0 then
3:         return 0
4:     end if
5:     for all j = 0; j < |T_i|; j++ do
6:         if x_ij = w then
7:             return t_ij
8:         end if
9:     end for
10:    return −1
11: end function
```

6.2 Function *SizeThresholdCalc*

The Algorithm 4 shows the pseudocode of the function *SizeThresholdCalc* which calculates the size threshold array. The function determines the size threshold of L_{p-1} by finding the greatest work-size in time function T_{p-1} whose execution time is less than τ (Lines 2-7). Then, it calculates σ_i, $i \in [0, p-2]$ where σ_i is the summation of σ_{i+1} with the greatest work-size in time function T_i whose execution time is less than τ (Lines 9-17).

It is supposed the all time functions in T are sorted in non-decreasing order of execution times.

6.3 Function *Cut*

The function *Cut* returns *TRUE* if work size w is greater than the passed size threshold σ (Algorithm 5).

6.4 Structure of matrix *Mem*

Two dimensional array *Mem* is used to save found solution for the visited nodes in the solution tree. It just memorizes the solution found on levels $\{L_1, \cdots, L_{p-2}\}$. Consider the memory cell $Mem[i][w]$. The memory cell

Algorithm 4 Algorithm Determining Size Thresholds

```
1: function SizeThresholdCalc(p, T, τ)
2:     size_max ← 0
3:     for all j = 0; GetTime(T_{p-1}, x_{(p-1) j}) < τ; j++ do
4:         if size_max < x_{(p-1) j} then
5:             size_max ← x_{(p-1) j}
6:         end if
7:     end for
8:     σ_{p-1} ← size_max
9:     for all i = p − 2; i ≥ 0; i−− do
10:        size_max ← 0
11:        for all j = 0; GetTime(T_j, x_ij) < τ; j++ do
12:            if size_max < x_ij then
13:                size_max ← x_ij
14:            end if
15:        end for
16:        σ_i ← σ_{i+1} + size_max
17:    end for
18:    return σ
19: end function
```

Algorithm 5 Algorithm Cutting Search Tree using the Size Threshold

```
1: function Cut(w, σ)
2:     if w > σ then
3:         return TRUE
4:     end if
5:     return FALSE
6: end function
```

saves the distribution of workload size w between processors $\{P_i, \cdots, P_{p-1}\}$. The saved information in $Mem[i][w]$ is composed of the following three fields:

- t_{mem}**:** $Mem[i][w].t_{mem}$ is the parallel execution time of the distribution found for workload size w at level L_i on processors $\{P_i, \cdots, P_{p-1}\}$. t_{mem} is initialized by constant $\_NE$. If $Mem[i][w].t_{mem}$ equals to $\_NE$, there is no saved solution for workload w at level L_i.
- idx_{last}**:** $Mem[i][w].idx_{last}$ is the index of the last data point in the time function T_i which has been examined. This data field helps *HPOPTA* to resume from where it was interrupted by *Backtrack*. In addition, idx_{last} is used to label a memory cell as *Finalized*. If idx_{last} equals to constant $\_FI$, the memory cell contains an optimal solution and therefore it is a *Finalized* memory cell.
- x_{mem}**:** $Mem[i][w].x_{mem}$ is the workload size assigned to P_i by the saved distribution.

6.5 Function *ReadMemory*

Algorithm 6 illustrates the function *ReadMemory*. Let w be workload size. First, $Mem[c][w]$ is accessed to read the last saved solution (Algorithm 6, Line 2). According to the retrieved values for t_{mem} and idx_{last}, the following cases are considered:

- **NOT_SOLUTION**: This case occurs when there is no saved execution time in memory (t_{mem} is $\_NE$), and the result has been finalized (idx_{last} is $\_FI$). It means that there is no solution for w on processor P_c (Lines 4-5).
- **SOLUTION**: This occurs when there is a finalized solution for workload w between processor $\{P_c, \cdots, P_{p-1}\}$. $D_{cur} = \{d_{cur}[0], \cdots, d_{cur}[c-1], \cdots, d_{cur}[p-1]\}$ contains the solution where $d_{cur}[i], \forall i \in [c, p-1]$ are retrieved from the memory. However, if the execution time of the saved solution is greater than τ ($t_{mem} > \tau$) the saved solution is ignored and NOT_SOLUTION is returned (Algorithm 6, Lines 6-15).
- **SOLUTION_RESUME**: This case is similar to the case SOLUTION, however in this case the solution is not finalized. The solution which has been already saved in Mem is first extracted into d_{cur} to be processed by *ProcessSolution*. In addition, it sets idx to idx_{last} retrieved from Mem. It makes *HPOPTA_Kernel* resume the process from the idx_{last}-th data point instead of starting from the beginning of T_c (Lines 16-23).
- **RESUME**: This condition happens when either there is already no solution to distribute the workload w between processor $\{P_c, \cdots, P_{p-1}\}$, and idx_{last} points to the index where the processing work was interrupted by backtracking, or idx_{last} points to the data point stored in $Mem[c][w].x_{mem}$ (Lines 24-27).

Algorithm 6 Algorithm Retrieving Solution from Memory

```
1: function ReadMemory(w, p, c, τ, T, D_cur, Mem, idx)
2:     ⟨t_mem, idx_last, x_mem⟩ ← Mem[c][w]
3:     if idx_last = _FI then
4:         if t_mem = _NE then
5:             return NOT_SOLUTION
6:         else
7:             if t_mem < τ then
8:                 d_cur[c] ← x_mem
9:                 d_cur[i] ← Mem[i][w − Σ_{j=c}^{i-1} d_cur[j]].x_mem,
10:                                                ∀i ∈ [c + 1, p − 2]
11:                d_cur[p − 1] ← w − Σ_{i=c}^{p-2} d_cur[i]
12:                return SOLUTION
13:            end if
14:            return NOT_SOLUTION
15:        end if
16:    else if idx_last ≠ _FI then
17:        if t_mem ≠ _NE ∧ t_mem < τ ∧ x_c idx_last ≠ x_mem then
18:            d_cur[c] ← x_mem
19:            d_cur[i] ← Mem[i][w − Σ_{j=c}^{i-1} d_cur[j]].x_mem, ∀i ∈ [c+1, p−2]
20:            d_cur[p − 1] ← w − Σ_{i=c}^{p-2} d_cur[i]
21:            idx ← idx_last
22:            return SOLUTION_RESUME
23:        end if
24:        if idx_last ≠ _NE then
25:            idx ← idx_last
26:            return RESUME
27:        end if
28:    end if
29:    return DUMMY
30: end function
```

6.6 Function *ProcessSolution*

The routine *ProcessSolution* (Algorithm 7) is invoked after finding a solution. At the beginning, it saves the solution into Mem (Lines 5-14). If m_{idx} was set to a positive integer, it means that some parts of solution from $P_{m_{idx}}$ to P_{p-1} was retrieved from Mem. If the execution time of the solution is less than current τ then D_{opt}, array σ and τ are updated (Lines 20-23). However, in case the execution time of current solution is equal to τ then D_{opt} will be updated by current solution if the number of idle processors (processors with allocated problem size of zero) in current solution is greater than that of D_{opt} (Lines 24-38).

6.7 Function *Save*

Algorithm 8 illustrates the function *Save* which is responsible to memorize the solution found for workload size w on

Algorithm 7 Algorithm Processing the Found Solution

```
1: function ProcessSolution(p, T, τ, σ, bk, D_cur, Mem, m_idx, D_opt)
2:     sum_size ← d_cur[p − 1]
3:     t_max ← GetTime(T_{p−1}, d_cur[p − 1])
4:     idx_max ← p − 1
5:     for i = p − 2, i ≥ 1, i − − do
6:         if GetTime(T_i, d_cur[i]) ≥ t_max then
7:             t_max ← GetTime(T_i, d_cur[i])
8:             idx_max ← i
9:         end if
10:        sum_size ← sum_size + d_cur[i]
11:        if sum_size ≠ 0 ∧ i < m_idx then
12:            Save(i, sum_size, t_max, d_cur[i], Mem)
13:        end if
14:    end for
15:    if GetTime(T_0, d_cur[0]) ≥ t_max then
16:        t_max ← GetTime(T_0, d_cur[0])
17:        idx_max ← 0
18:    end if
19:    bk ← idx_max
20:    if τ > t_max then
21:        τ ← t_max
22:        D_opt ← D_cur
23:        σ ← SizeThresholdCalc(p, T, τ)
24:    else if τ = t_max then
25:        proc_cur ← 0
26:        proc_last ← 0
27:        for i = 0, i < p, i + + do
28:            if d_cur[i] ≠ 0 then
29:                proc_cur + +
30:            end if
31:            if d_opt[i] ≠ 0 then
32:                proc_last + +
33:            end if
34:        end for
35:        if proc_cur < proc_last then
36:            D_opt ← D_cur
37:        end if
38:    end if
39: end function
```

processors $P_i, \cdots, P_{p-1}$. t_{max} is parallel execution time of the solution and d is the size of workload assigned to P_i. The function will not update a memory cell if the newly found distribution has greater parallel execution time than the execution time of the saved solution.

Algorithm 8 Algorithm Storing Solution into Matrix Mem

```
1: function Save(i, w, t_max, d, Mem)
2:     if Mem[i][w].t_mem < t_mem then
3:         Mem[i][w] ← ⟨t_max, −, d⟩
4:     end if
5: end function
```

6.8 Function *Backtrack*

Algorithm 9 shows the pseudocode of *Backtrack* which is called by *HPOPTA_Kernel* to make decision if the process backtracks to the upper node or not. The function returns *TRUE* in case a backtracking should be done. In addition, this function performs memory finalization.

6.9 Function *MakeFinal*

Algorithm 10 illustrates the function *MakeFinal* which is responsible to finalize the memory cell mem. It sets idx_{last} to the constant value $\_FI$. The finalized memory cell contains the optimal distribution.

7 Correctness Proof of *HPOPTA*

Proposition. *5.1.* *The algorithm* HPOPTA *always returns a distribution of the workload of size n between p heterogeneous processors that minimizes its parallel execution time.*

Algorithm 9 Algorithm Implementing Backtracking and Matrix Mem Cell Finalization

```
1: function Backtrack(w, c, bk, idx, t_cur, τ, Mem, isMem)
2:     if bk < c then
3:         if isMem = TRUE then
4:             return TRUE
5:         end if
6:         if t_cur = τ then
7:             Mem[c][w].idx_last ← _FI
8:         else
9:             Mem[c][w].last_index ← idx
10:        end if
11:        return TRUE
12:    else if bk = c then
13:        bk ← NULL
14:        Mem[c][w].idx_last ← _FI
15:        return TRUE
16:    else
17:        bk ← NULL
18:        return FALSE
19:    end if
20: end function
```

Algorithm 10 Algorithm Finalizing a Matrix Mem cell

```
1: function MakeFinal(mem)
2:     mem.idx_last ← _FI
3: end function
```

Proof. To find a distribution of workload n between processors $\{P_0, \cdots, P_{p-1}\}$ that minimizes its execution time, the straightforward approach is to examine all possible distributions using the full search tree. This approach has however exponential complexity. Instead, *HPOPTA* only builds and explores a small fraction of the full search tree. To achieve this, it applies two specific operations, *Cut* and *Backtrack*, that remove subtrees of the full search tree without their construction and exploration. Therefore, the correctness of *HPOPTA* will be proved if we show that no subtree removed by *HPOPTA* from consideration contains a solution, superior to the one eventually returned by *HPOPTA*.

Cut: During execution of *HPOPTA*, this operation removes a subtree, growing from a node at level L_i, only if the workload w associated with this node is greater than σ_i. Remember that σ_i represents the maximum workload that can be executed in parallel by processors $\{P_i, \cdots, P_{p-1}\}$ faster than in τ time units, and τ is the execution time of the fastest solution found so far. Therefore, the parallel execution time of any solution in the removed subtree cannot be less than τ and hence less than the time of the globally fastest solution.

Backtrack: This operation is only applied when during its execution *HPOPTA* finds a solution $\{x_0, x_1, \cdots, x_k, \cdots, x_{p-1}\}$ such that $\max_{i=0}^{p-1} t_i(x_i) = t_k(x_k) = \tau$. We know two facts:

1) The parallel execution time of any solution in the subtree with the root x_k cannot be less than τ. Therefore, this subtree can be ignored.
2) *HPOPTA* arranges nodes at each level of the search tree in non-decreasing order of their execution times. Therefore, all nodes at level L_k that follow the node x_k will have execution times greater than or equal to $t_k(x_k)$.

From these facts we can conclude that construction and examination of the subtrees, which would grow from the node x_k and the following nodes at level L_k of the search

tree, will not result in a distribution with the execution time less than τ and can be ignored by backtracking to level L_{k-1}. *End of Proof.*

8 COMPLEXITY OF *HPOPTA*

Lemma 8.1. *The complexity of* HPOPTA_Kernel *is* $O(m^3 \times p^3)$.

Proof. HPOPTA_Kernel expands a search tree recursively. Thus, its time complexity can be expressed in terms of the number of recursive invocations of *HPOPTA_Kernel*. We formulate the number of recursive calls in each level of the search tree using a trivial example.

Consider a workload size w is executed on a platform consisting of 5 heterogeneous processors ($p = 5$). Suppose the time function of each processor contains 2 data points ($m = 2$) where $t(x) = \{(\Delta x, t(\Delta x)), (2\Delta x, t(2\Delta x))\}$, and $\Delta x \in \mathbb{N}$ is the minimum granularity of workload sizes in time functions. It should be mentioned that there is no assumption about the value of Δx and therefore it can be used without loss of generality. For sake of simplicity, we assume that the execution time increases as worksize increases in a time function. That is, the data points in time functions are visited in increasing order of work-size. This assumption does not make the proof less general. However, it makes finding the formula for complexity less difficult.

Figure 4 shows the search tree of this example. Since we are looking for an upper bound for time complexity of *HPOPTA_Kernel*, we consider w greater than $8\Delta x$, which is the maximum workload size subtracted from w in the Figure. For the sake of simplicity, only *Save* technique is considered. Other optimizations, *Cut* and *Backtrack*, are not applied. In the Figure, nodes highlighted in red have already been expanded in the same level and the solutions for them are retrieved from the matrix Mem instead of node expansions.

The number of recursive calls in each level of tree can be calculated as follows:

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p-1 \\ C\#(p-2) \times (m+1) & L = p-1 \end{cases}$$

where L represents level number in the tree.

Therefore, $C\#(L)$ is equal to:

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p-1 \\ m^2 \times p - 2 \times m^2 + m \times p - m + 1 & L = p-1 \end{cases}$$

That is, the total number of recursive calls of *HPOPTA_Kernel* is $\sum_{L=0}^{p-1}(C\#(L))$ which has order of $O(m \times p^2 + m^2 \times p)$.

In addition, the number of nodes in each level whose results are retrieved from matrix Mem is equal to:

$$\begin{aligned} \text{Memory\#(L)} &= (C\#(L-1)-1) \times m \\ &= (m^2) \times (L-1), \\ & 1 \leq L \leq p-2 \end{aligned}$$

Solutions can be found by using the memory. To retrieve saved results, the function *ReadMemory* is required to access up to $p-2$ elements from Mem. That is, the complexity of *ReadMemory* is $O(p)$. We know *HPOPTA_Kernel* accesses the matrix Mem for levels 1 to $p-2$. Therefore, the cost of all *ReadMemory* invocations to find solutions is equal to $\sum_{L=1}^{p-2}(Memory\#(L) \times O(p))$, which is equal to $O(m^2 \times p^3)$.

In addition, there are solutions found in the last level of the tree (For instance level L_4 in the Figure 4). For these solutions, *HPOPTA_Kernel* accesses $t_{p-1}(x)$ with the time complexity of $O(m)$ to find the execution time of the given workload to P_{p-1}. The number of nodes in the level L_{p-1} is equal to $C\#(p-1)$ nodes. That is, the cost of finding solutions in last level of tree is $O(m^3 \times p)$.

Once a solution is found, *ProcessSolution* is invoked and it has complexity of $O(m \times p)$. In the worst case, it is invoked in each leaf of the tree, either after each call of *ReadMemory* or after finding a solution in level L_{p-1}. Therefore, total complexity associated with all *ProcessSolution* calls is bound to $(C\#(p-1) + \sum_{L=1}^{p-2} Memory\#(L)) \times O(m \times p) = O(m^3 \times p^3)$.

The worst time complexity of *HPOPTA_Kernel* can therefore can be summarized as follows:

Complexity(*HPOPTA_Kernel*) =
O(recursive calls of *HPOPTA_Kernel*) + O(*ReadMemory* calls)+
O(finding solutions in L_{p-1}) + O(*ProcessSolution* calls).

which equals:

$$\begin{aligned} \text{Complexity(}&\textit{HPOPTA\_Kernel}\text{)} = \\ & O(m \times p^2 + m^2 \times p) + \\ & O(m^2 \times p^3) + \\ & O(m^3 \times p) + O(m^3 \times p^3) \\ & = O(m^3 \times p^3). \end{aligned}$$

Proposition. 5.2. *The complexity of* HPOPTA *is* $O(m^3 \times p^3)$.

Proof. HPOPTA consists of following main steps:

- **Sorting:** There are p discrete time functions with cardinality of m. The complexity to sort all of them in the non-decreasing order of execution times is $O(p \times m \times \log_2 m)$.
- **Finding load–equal distribution and initialization of time threshold:** This step has complexity $O(p)$.
- **Finding size thresholds:** To find size threshold for a given level L_i, $i \in [0, p-1]$, all data points, existing in $t_i(x)$, with execution times less than or equal with τ should be examined. This has complexity of $O(m)$. Therefore, finding p size thresholds has a complexity of $O(p \times m)$.
- **Memory initialization:** In this step, all $(n+1) \times (p-2)$ cells of Mem are initialized with the complexity of $O(n \times p)$.
- **Kernel invocation:** According to the lemma 8.1, the complexity of this step is $O(m^3 \times p^3)$.

Therefore, the time complexity of *HPOPTA* equals the summation of all these steps, which is equal to $O(m^3 \times p^3)$.

End of Proof.

Proposition. 5.3. *The total memory used by the algorithm is* $O(p \times (m+n))$.

Proof. HPOPTA uses memory to store following information:

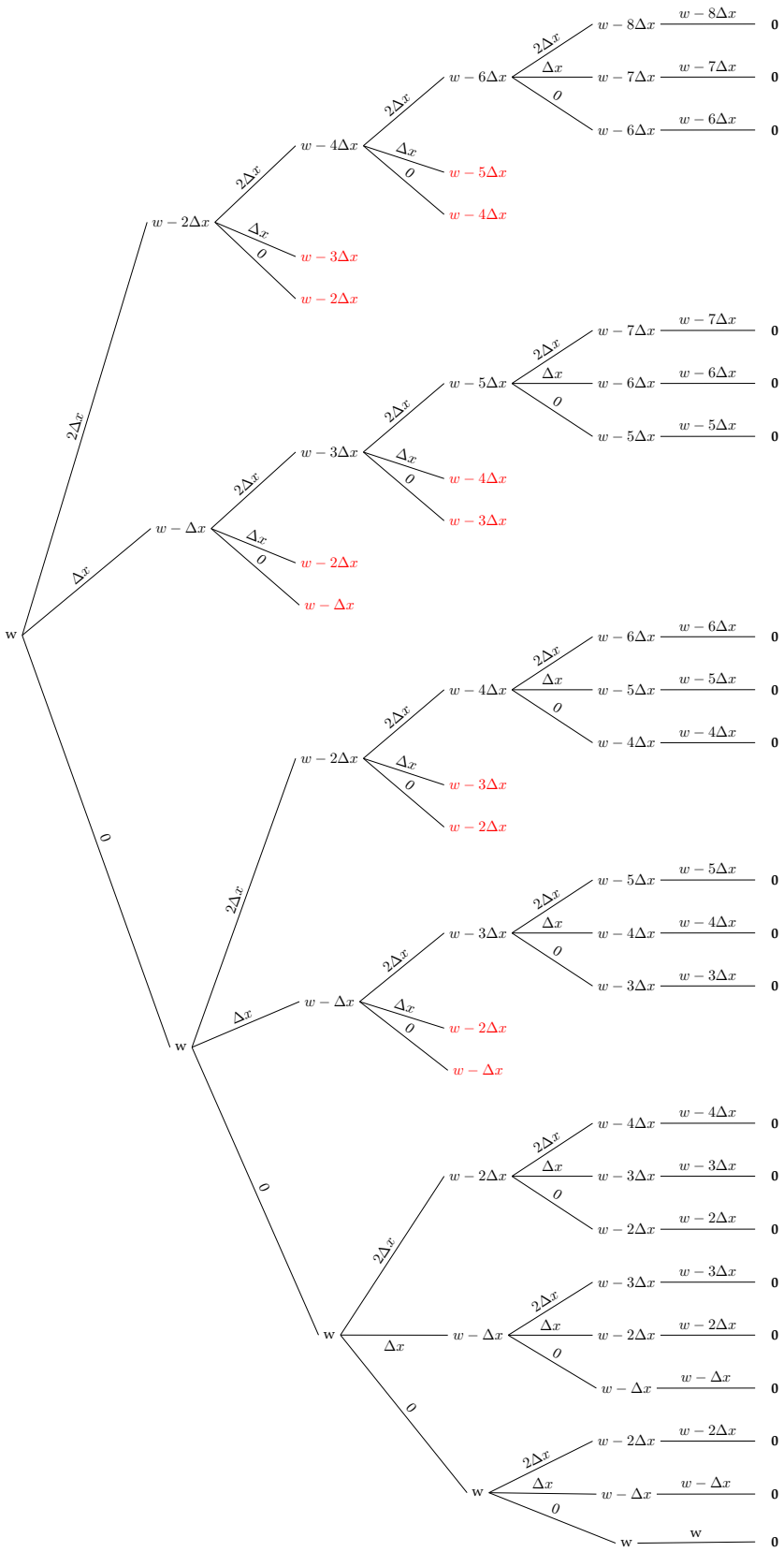

Fig. 4. The execution of *HPOPTA* for a sample set of time functions ($p = 5$), each contains 2 data points. The memorization technique is only considered to reduce the full search space of solutions. The other optimizations, time threshold, size threshold, and backtracking, are not applied.

- **time functions:** There are p discrete time functions with cardinality of m stored in $p \times m$ cells.
- D_{cur}**:** This array stores the workload sizes assigned to each processor by the current solution. That is, this is an array of size p cells.
- D_{opt}**:** This array stores the workload sizes assigned to each processor by the optimal solution found so far. That is, this is an array of size p cells.
- **Mem:** This is a matrix consisting of $(p-2) \times (n+1)$ cells.

Thus, total memory cells used by *HPOPTA* is equal to $m \times p + 2 \times p + (p-2) \times (n+1) \approxeq O(p \times (m+n))$.

End of Proof.

ACKNOWLEDGEMENTS

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

REFERENCES

[1] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2017.

[2] A. Lastovetsky and R. Reddy, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2017.

Hamidreza Khaleghzadeh is a PhD researcher at Heterogeneous Computing Laboratory at the School of Computer Science, University College Dublin. He got his BSc and MSc degrees in Computer Engineering (software) in 2007 and 2011, respectively. He ranked as a 2nd position holder in his MS program. His main research interests include performance and energy consumption optimization in massively heterogeneous systems, high performance heterogeneous systems, energy efficiency, and parallel/distributed computing.

Ravi Reddy Manumachu received a B.Tech degree from I.I.T, Madras in 1997 and a PhD degree from the School of Computer Science, University College Dublin in 2005. His main research interests include high performance heterogeneous computing, high performance linear algebra, parallel computational fluid dynamics and finite element analysis.

Alexey Lastovetsky received a PhD degree from the Moscow Aviation Institute in 1986, and a Doctor of Science degree from the Russian Academy of Sciences in 1997. His main research interests include algorithms, models, and programming tools for high performance heterogeneous computing. He has published over a hundred technical papers in refereed journals, edited books, and international conferences. He authored the monographs Parallel computing on heterogeneous networks (Wiley, 2003) and High performance heterogeneous computing (Wiley, 2009).