MultiSphere: Massively Parallel Tree Search for Large Sphere Decoders

Konstantinos Nikitopoulos, Daniil Chatzipanagiotis, Chathura Jayawardena, and Rahim Tafazolli
Institute for Communication Systems (ICS)
Home of 5G Innovation Center
University of Surrey, Guildford, UK
Email: {k.nikitopoulos, dc00121, c.jayawardena, r.tafazolli}@surrey.ac.uk

***Abstract*—This work introduces MultiSphere, a method to massively parallelize the tree search of large sphere decoders in a nearly-independent manner, without compromising their maximum-likelihood performance, and by keeping the overall processing complexity at the levels of highly-optimized sequential sphere decoders. MultiSphere employs a novel sphere decoder tree partitioning which can adjust to the transmission channel with a small latency overhead. It also utilizes a new method to distribute nodes to parallel sphere decoders and a new tree traversal and enumeration strategy which minimize redundant computations despite the nearly-independent parallel processing of the subtrees. For an 8×8 MIMO spatially multiplexed system with 16-QAM modulation and 32 processing elements MultiSphere can achieve a latency reduction of more than an order of magnitude, approaching the processing latency of linear detection methods, while its overall complexity can be even smaller than the complexity of well-known sequential sphere decoders. For 8×8 MIMO systems, MultiSphere's sphere decoder tree partitioning method can achieve the processing latency of other partitioning schemes by using half of the processing elements. In addition, it is shown that for a multi-carrier system with 64 subcarriers, when performing sequential detection across subcarriers and using MultiSphere with 8 processing elements to parallelize detection, a smaller processing latency is achieved than when parallelizing the detection process by using a single processing element per subcarrier (64 in total).**

***Index Terms*—Sphere Decoding, Parallel Processing, Large Multiple-Input–Multiple-Output (MIMO), Lattice Search.**

I. Introduction

The ever-increasing need for wireless capacity has recently triggered a paradigm shift from orthogonal to non-orthogonal signal transmission. By intentionally transmitting non-orthogonal, and therefore, mutually interfering information streams, substantial capacity gains can be achieved. However, to deliver the theoretical gains in practice, these streams must be optimally demultiplexed. Sphere decoding is a well-known technique that substantially reduces the complexity for optimally (in the maximum likelihood (ML) sense) detecting mutually interfering information streams, by translating the ML detection problem into a tree search ([1], [2], [3]). While the throughput gains of sphere decoding increase with the number of the mutually interfering information streams the corresponding processing requirements increase exponentially, exceeding the processing capabilities of traditional processors. These processing requirements, in combination with the reaching of a plateau in the speed of traditional microprocessors [4] prevent traditional systems from supporting large numbers of mutually interfering streams and, therefore, from scaling the achievable throughput gains.

At the same time, emerging system-on-chip architectures promise tens or even hundreds of cores per chip [5]; something which is already feasible in graphics processing units (GPUs). In the presence of such multiple processing element (PE) architectures the complexity problem translates into how to efficiently utilize the available PEs or equivalently, into how to efficiently parallelize the processing load. Parallelizing the sphere decoder (SD) is a challenging task since its computational efficiency is determined by the ability to prune (i.e., exclude nodes from the tree search) large parts of the tree at an early stage of the tree search without compromising its algorithmic optimality. As a result, typical computationally efficient solutions are sequential ([2], [3]).

An "ideal" SD parallelization method should be:

- *scalable* and able to consistently reduce latency when increasing the number of PEs,
- *complexity efficient* and should not (substantially) increase the overall processing load when increasing the number of PEs,
- *nearly "embarrassingly parallel"* and therefore minimize dependencies and communication overhead which introduce latency and can moderate if not obliterate the scalability and efficiency of parallelization [6],
- *adjustable to the transmission conditions* so that they can efficiently allocate the processing power (PEs) to minimize processing latency,
- *applicable to all kinds of SDs*, including breadth-first and depth-first SDs, as well as exact (guaranteeing the ML solutions) and approximate SDs.

With the above characteristics the processing latency reduction can be maximized for a given transmission scenario. In addition, the latency gains can be efficiently translated into power gains (by hardware clock reduction), and the system design becomes very flexible since the parallel sub-tasks can even run on individual processing blocks. Such a parallelization method is also transparent to the choice of the implementation platform. In Many-Processor systems on Chips (MPSoCs), for example, a PE can be a different processor, in FPGA designs it can be a specifically allocated part of the FPGA chip and

in GPU implementations the PE can be a separate thread.

Many SD implementations involve parallelism, but without meeting the characteristics discussed above. For example, both depth-first [2], [7] and breadth-first [8] SDs perform several Euclidean distance calculations in parallel, at each level of the SD tree, exploiting a limited level of data parallelism. However, after performing a set of parallel computations, before the next set is processed in parallel, the sorting operations required introduce significant dependencies. As a result, this kind of parallelization is highly dependent on the specific hardware realization, it is not flexible, and it cannot be efficiently used for decreasing the latency requirements of large SDs.

In GPU implementations, Khairy et al. [9] concurrently run multiple, low-dimensional (4×4) SDs but without parallelizing each SD. Wu et al. [10] and Josza et al. [11] attempt to also parallelize the low-dimensional MIMO detection process, on GPUs. However, Wu et al. use a Trellis-decoder-like approximation of the SD which is not efficient for dense modulations and large MIMO systems, and Jósza et al. perform aggressive and nearly exhaustive parallel search of multiple subtrees without accounting for the overall complexity and by exhaustively trying different partitioning configurations. As a result, their approach is not appropriate for MPSoC or FPGA implementations and lacks theoretical reasoning.

Yang et al. [12], [13] propose a multi-core depth-first architecture for parallel high-dimensional SDs. To parallelize their SD tree search, the SD tree is partitioned in subtrees which consist of only one node on the higher layers, and all possible nodes at the lower layers of the tree. Their SD partitioning starts by first allocating to subtrees all the nodes of the higher level, and if there are still available PEs each of the subtrees is further partitioned using the same principles. This SD partitioning is very practical in terms of implementation, but, it cannot adjust to the transmission conditions. In addition, to avoid visiting a node twice and control the overall complexity, Yang et al. use an interconnection network which determines which of the nodes will be processed from each PE and it distributes the most promising solution from each of the subtrees. This reduces the flexibility of the approach, and therefore its efficiency when applied to implementations that require nearly independent parallel processing, as in the case of GPU implementations or implementations on individual processing blocks.

The fixed complexity SD [14] sacrifices the ML optimality for acquiring very good parallelization properties. However, to efficiently run such an SD in parallel, the available number of PEs should be a multiple of the order of the transmitted constellation. In addition, the way it decides which tree paths to run in parallel, is pre-defined and cannot be adjusted to the transmission conditions. These weaknesses can be rectified by the methods proposed in this paper.

This work proposes MultiSphere, a method to efficiently split the SD tree search in nearly-embarrassingly parallel subsearches. MultiSphere is scalable to large numbers of PEs, and preserves ML optimality with a small complexity overhead. In addition, its tree partitioning can adjust to the transmission channel, reducing substantially the processing latency for a given number of PEs. In particular, MultiSphere consists of:

- *A novel SD partitioning scheme adjustable to the transmission channel*: MultiSphere focuses the available processing power (i.e., the PEs) on the paths of the SD tree that are more likely to include the transmitted vector. To achieve this MultiSphere's partitioning can adjust to the transmission channel or its statistics, and can take place either offline (based on the channel statistics) or "on-the-fly" any time the transmission channel changes.
- *A novel complexity-efficient method to allocate symbols to the parallel subtrees* and allows nearly-embarrassingly parallel processing while minimizing redundant calculations across the parallel subprocesses.
- *A novel tree traversal and enumeration strategy* that minimizes unnecessary Euclidean distance calculations, and in contrast to existing approaches ([3], [7], [15]) that apply only to sequential SDs, it can also apply to both sequential and parallel ones.

MultiSphere applies to any kind of SD. Still, this work focuses on parallel depth-first SDs that can guarantee the ML solution ([2], [3]). This is probably the most challenging SD case since such SDs owe their efficiency to their strictly-sequential tree traversal and pruning. In addition, while MultiSphere applies to any system allowing mutually interfering information streams this work focuses on MIMO systems.

II. Sphere Decoding for MIMO Systems

For a spatially multiplexed MIMO system consisting of n_t transmit and n_r receive antennae the received signal vector is $\mathbf{y} = \mathbf{Hs} + \mathbf{w}$, where $\mathbf{H}$ is the $n_r \times n_t$ MIMO channel matrix, $\mathbf{s}$ is the transmitted symbol vector whose elements belong to a constellation $\mathcal{O}$ of size $|\mathcal{O}|$ and $\mathbf{w}$ is the additive white Gaussian noise vector. By QR decomposing the MIMO channel matrix as $\mathbf{H} = \mathbf{QR}$ the ML problem is translated into finding [1], [2], [3]

$$\hat{\mathbf{s}} = \arg\min_{\mathbf{s}\in\mathcal{O}^{n_t}} \|\widetilde{\mathbf{y}} - \mathbf{Rs}\|^2. \tag{1}$$

with R_{ij} being the elements of $\mathbf{R}$, and $\widetilde{\mathbf{y}} = \mathbf{Q}^*\mathbf{y}$. Since $\mathbf{R}$ is an upper triangular matrix, finding the ML solution can be transformed to a search tree of height n_t and branching factor $|\mathcal{O}|$. Each node at a level l can be characterized by its *partial symbol vector* $\mathbf{s}^{(l)} = [s_l, s_{l+1}, \ldots, s_{n_t}]$ which gives the path from the root to these nodes, as well as from its partial Euclidean distance (PD) which can be calculated recursively as $d(\mathbf{s}^{(l)}) = d(\mathbf{s}^{(l+1)}) + c(\mathbf{s}^{(l)})$ where $c(\mathbf{s}^{(l)})$ is the non-negative cost assigned to each branch,

$$c(\mathbf{s}^{(l)}) = \left| \widetilde{y}_l - \sum_{j=l}^{n_t} R_{lj} s_j \right|^2. \tag{2}$$

Then, the ML problem is translated into finding the leaf-node with the minimum $d(\mathbf{s}^{(1)})$. For depth-first SDs with radius update and Schnorr-Euchner enumeration [2], [3], the initial squared radius r^2 is set to be infinite. Any time a leaf

node is reached with $d(\mathbf{s}^{(1)}) < r^2$, r^2 is updated to $d(\mathbf{s}^{(1)})$. Upon meeting a node $\mathbf{s}^{(l)}$, if $d(\mathbf{s}^{(l)}) \geq r^2$, this node, its children nodes and its siblings with all their descendants are excluded from the tree search (i.e., they are pruned). Following the Schnorr-Euchner tree traversal [16] when expanding a node, the nodes are visited in ascending order of their PDs. Since depth-first SDs with radius update and Schnorr-Euchner enumeration have been shown to be very efficient in practice [2], [3] and capable of delivering the ML solutions, this is the structure the we will adopt for all our parallel SDs.

III. MULTISPHERE DESIGN

MultiSphere consists of a new *SD tree partitioning method*, which adjusts to the transmission channel without compromising the ML optimality (see Section III-A). The partitioning can take place offline, based on the average channel characteristics, or "on-the-fly", when the transmission channel changes after each QR decomposition. This adds preprocessing latency to that of the QR decomposition. However, the partitioning latency scales linearly with n_t in contrast to the QR decomposition latency which scales almost cubicly with the number of transmit antennae. After SD partitioning, MultiSphere applies a new *symbol-to-subtree allocation method* (see Section III-B) which efficiently maps nodes to PEs without introducing dependencies, in contrast to other schemes [12], and minimizes the number of redundant calculations across PEs. Each PE performs depth-first subtree traversal with Schnorr-Euchner enumeration [16], according to which, nodes are visited in ascending order of their PDs. Several approaches have been proposed [2], [3], to avoid exhaustively calculating and sorting the PDs. However, they are not applicable to MultiSphere since their ordering is sequential (for finding the k^{th} smallest PD, the $(k-1)^{th}$ smallest PD must be first found starting from $k = 1$). In Section III-C a new *tree traversal and enumeration method* is introduced for meeting MultiSphere's needs.

MultiSphere runs the parallel SDs in a nearly independent form. They interact only once, after they have all reached the first leaf node. Then, the r^2 of each subtree is replaced by the value of the leaf node with the minimum PD across all parallel SDs. The search is terminated when all parallel trees have been searched. Then, the detection output is the leaf node with the minimum PD across all subtrees and the overall processing latency is determined by the slowest parallel SD.

A. MultiSphere's SD Tree partitioning

MultiSphere's SD partitioning consists of the *seeds identification*, which finds the N_{PE} most promising paths (seeds) to constitute the correct solution (i.e., to be the transmitted vector) with N_{PE} being the number of available PEs, as well as of the *subtrees construction*, which assembles subtrees around the seeds so that their union forms the original SD tree.

1) MultiSphere's Seeds Identification: The *relative position vector* (RPV) $\mathbf{m}$ describes a tree path by means of the ordered (in terms of PDs) position of its nodes to the received observable. In particular, if the l^{th} element of $\mathbf{m}$ equals k, then, for the corresponding path, its node at level l is the k^{th} closest node to the received $\widetilde{y}_l$. If, for example, $\mathbf{m} = [1,2,3]^T$ the path consists of the node with the third smallest PD at the highest level of the tree, its child with the second smallest PD at the second level of the tree, and its child with the smallest PD at the lowest level of the SD tree.

Finding the exact probability for each path to include the correct solution is clearly a non-trivial task and would require difficult integrations with no obvious closed-form solutions. In order to simplify the task we use a heuristic metric of promise (MoP) $\mathcal{M}$ of probabilistic reasoning. Calculating more accurate and efficient MoPs is part of our future research. The proposed MoP is equal to the PD of a path in the nearly noiseless case. In the absence of noise, the correct solution would be the one with $\mathbf{m} = [1,1,1]^T$ and the $N_{PE} - 1$ paths with the smallest MoPs would be the ones closest to the $\mathbf{m} = [1,1,1]^T$. Similarly, in the presence of noise, we assume that the N_{PE} most promising paths are the ones with the N_{PE} smallest MoPs. These MoPs are recursively calculated as

$$\mathcal{M}(s_l) = \mathcal{M}(s_{(l+1)}) + |R_{ll}|^2 \left| s_l^{(c)} - s_l \right|^2 \qquad (3)$$

with $\mathcal{M}(s_{(n_t+1)}) = 0$ and $s_l^{(c)}$ being the symbol at level l which is closest to the received point. Equation 3 shows that these MoPs are not a function of the actual symbols but of the ordered distances between transmitted symbols which are known in advance and can be pre-calculated. Such a metric also allows representing the MoPs by means of RPVs. Also, Eq. 3 shows that in order to calculate the N_{PE} most promising paths $|\mathcal{O}|$ complex multiplications are required per level l. Since its recursive structure is similar to the SD, but with branches of equal weight, the most promising nodes are found in K-Best manner with $K = N_{PE}$, resulting in latency requirements of the order of n_t. Alternatively, the processing can take place offline, by replacing R_{ll} with its expected value. In all cases, since the actual $s_l^{(c)}$ is not known when calculating the seeds, it is assumed to be an inner constellation symbol.

2) MultiSphere's Subtrees Construction: After calculating the N_{PE} seeds with RPVs $\mathbf{m}_i$ ($i = 1, ..., N_{PE}$), each of them is used to construct a corresponding subtree T_i so that the union of all subtrees forms the original SD tree. The process is designed in a way that PEs can independently construct their subtrees in parallel in order to minimize the latency of the procedure. For convenience the seeds $\mathbf{m}_i$ are sorted in ascending order of indices at level n_t; seeds with the same index at n_t are in ascending order of indices at level $n_t - 1$ and so on, e.g., $\mathbf{m}_1 = [1,1,1]^T$, $\mathbf{m}_2 = [1,1,2]^T$, $\mathbf{m}_3 = [1,2,2]^T$, $\mathbf{m}_4 = [1,1,3]^T$. For each $\mathbf{m}_i$ the process is initiated at level $l = n_t$ by checking if the element $m_{i,l}$ is unique across i at level l (i.e., if $m_{i,l} \neq m_{k,l}$ for $k \neq i$). If so, this node and its descendants are included to subtree T_i. If $m_{i,l}$ is maximum across i in l its sibling nodes to the right with indices up to $|\mathcal{O}|$ and their descendants are also included to T_i. If $m_{i,l}$ is not unique across i the indices k of the non-unique seeds are saved in a (per level) buffer $\mathcal{B}_l$ as they will be needed later. Traversing down the tree ($l \leftarrow l-1$), the indices of non-unique seeds at l from $\mathcal{B}_{(l+1)}$ are saved in $\mathcal{B}_l$ until the paths of the

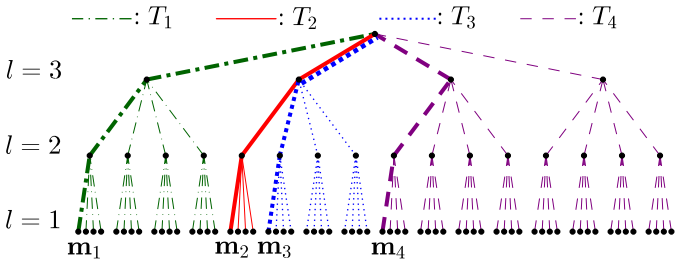

Fig. 1. Construction of subtrees for seeds $\mathbf{m}_1 = [1,1,1]^T$, $\mathbf{m}_2 = [1,1,2]^T$, $\mathbf{m}_3 = [1,2,2]^T$, $\mathbf{m}_4 = [1,1,3]^T$ (bold lines) into the subtrees T_1 (dashed-dotted line), T_2 (solid line), T_3 (dotted line) and T_4 (dashed line). Their union it the full SD tree. Nodes may appear in several subtrees.

non-unique seeds "split" and a node is found to be unique to seed i. Then it is examined if $m_{i,l}$ is the maximum across i amongst the non-unique seeds in $\mathcal{B}_{(l+1)}$. If it is not, the node, its descendants and ancestors are included to T_i. If it is, the node, its siblings to the right with indices up to $|\mathcal{O}|$ and all the descendants are included to T_i and walking up the tree in a similar way, the appropriate nodes are included by comparing with the maximum in $\mathcal{B}_{(l+1)}$ until the level n_t is reached. The pseudocode for this process is given in Algorithm 1. The algorithm will, at the worst case, go to level $l = 1$ of the tree to find a unique $m_{i,1}$ value, and if $m_{i,1}$ is the maximum at $l = 1$ amongst seeds in $\mathcal{B}_{(l+1)}$ it will have to go up to level $l = n_t$ to allocate sibling nodes; resulting in a latency of $2n_t$. However, in practice, seeds tend to be unique in the higher levels of the tree, resulting in a smaller average latency.

B. MultiSphere's Symbol-to-Subtree Allocation Method

MultiSphere's tree partitioning gives the nodes to be processed by each parallel SD as a function of their ordered distance. In the example of Fig. 1, subtree T_3 will consist of the 2^{nd} closest symbol at $l = 3$, its children which are 2^{nd}, 3^{rd} and 4^{th} closest at $l = 2$ and all their children at $l = 1$. Finding the actual symbols would require exhaustive PD calculations and sorting of the corresponding nodes multiple times across the parallel SDs. To avoid these redundant calculations, MultiSphere uses an approximate predefined visiting order, based on calculating *minimum Euclidean distances* depending on the relative position of the received point and the constellation geometry. In addition, it uses a symbol mapping of *two-dimensional zigzag coordinates*. In one-dimensional symbol constellations, the sorted order of the symbols in terms of their distance to the received point can be easily found in a zigzag manner [2], [3] after finding the closest constellation symbol $s_l^{(c)}$ to the "equivalent (in the constellation domain) received point". The equivalent received point is

$$\overline{y}_l = \left(\widetilde{y}_l - \sum_{j=l+1}^{n_t} R_{lj} s_j \right) R_{ll}^{-1}. \tag{4}$$

Using the zigzag concept each symbol in a two-dimensional constellation can be mapped in terms of its zigzag coordinates (zz_x, zz_y), as shown in Fig. 2. MultiSphere's preordering is

Algorithm 1 Construction of subtrees

Inputs: seeds $\mathbf{m}$, n_t, $|\mathcal{O}|$
$l \leftarrow n_t$ where l denotes the current level
$\mathcal{B}_{(l+1)} \leftarrow \{1, ..., N_{PE}\}$ $\mathcal{B}_l$ is a buffer with indices k of non-unique seeds at l ($m_{il} = m_{kl}$). Initialize $\mathcal{B}_{(n_t+1)}$ with all seeds
if $m_{i,l}$ is not unique ($m_{il} = m_{kl}$ for $k \neq i$, $k \in \mathcal{B}_{(l+1)}$) **then**
  save indices of non-unique seeds in buffer $\mathcal{B}_l$
  $l \leftarrow l - 1$
  go to step 4
else
  if $m_{i,l} < \max_{k \in \mathcal{B}_{(l+1)}} m_{k,l}$ **then**
    $limit(l) \leftarrow m_{(i+1),l} - 1$
    include nodes at l with indices $m_{i,l} \leq j \leq limit(l)$, all their descendants and ancestors to subtree T_i
  else
    $limit(l) \leftarrow |\mathcal{O}|$
    include nodes at l with indices $m_{i,l} \leq j \leq limit(l)$ and all their descendants to subtree T_i
    $l \leftarrow l + 1$
    while $l \leq n_t$ **do**
      include node at l with index $m_{i,l}$ to subtree T_i
      if $m_{i,l} < \max_{k \in \mathcal{B}_{(l+1)}} m_{k,l}$ **then**
        $limit(l) \leftarrow m_{(i+1),l} - 1$
        include nodes at l with indices $m_{i,l} < j \leq limit(l)$, their descendants and ancestors to subtree T_i
        break
      else
        $limit(l) \leftarrow |\mathcal{O}|$
        include nodes at l with indices $m_{i,l} \leq j \leq limit(l)$, and their descendants to subtree T_i
      end if
      $l \leftarrow l + 1$
    end while
  end if
end if
Output: T_i

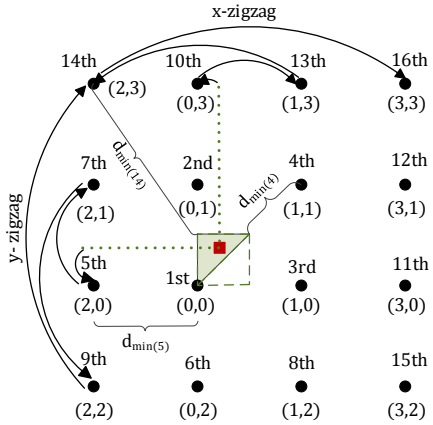

Fig. 2. MultiSphere's predefined order example for 16-QAM.

based on the relative position of $\overline{y}_l$ (square point in Fig. 2) to $s_l^{(c)}$ and consists of two cases, depending on whether $|Im\{\overline{y}_l - s_l^{(c)}\}|$ is larger or smaller than $|Re\{\overline{y}_l - s_l^{(c)}\}|$. For each case the relative position of the received point around $s_l^{(c)}$ is known (shadowed triangle in Fig. 2). Thus, a minimum Euclidean distance d_{min} from $\overline{y}_l$ to any constellation point can be calculated. Then, MultiSphere's predefined order, approximates the actual order, and is such that the constellation

symbols are in ascending order of their d_{min}. To calculate the d_{min} values it is assumed that $s_l^{(c)}$ is an inner constellation symbol. Since the sequence is stored in zigzag coordinates, mapping is feasible even if $s_l^{(c)}$ is an outer constellation symbol. Then, d_{min} is still a valid lower limit of the corresponding Euclidean distance, since the zigzag from $s_l^{(c)}$ will point to a symbol which is even further than what was initially assumed.

C. MultiSphere's Tree Traversal and Enumeration

When expanding a node, MultiSphere's parallel SDs visit children nodes in ascending order of their PDs (see Section II). To avoid calculating and sorting all PDs at each level, enumeration methods have been proposed [2], [3] which, as discussed, are not applicable to MultiSphere since their ordering is sequential. MultiSphere first checks the minimum d_{min} of the symbols that need to be expanded. If it is larger than r^2, the descendants of this node, as well as its siblings and all their descendants can be safely pruned. If not, from the set of potential symbols to be visited at this specific tree level, for each existing zz_x zigzag coordinate, the symbols with the minimum zz_y are identified. This results in a subset of at most $\sqrt{|\mathcal{O}|}$ symbols with unique zz_x coordinates. From this subset, the PD of the symbol with the minimum zz_x value having $zz_y = 0$ is calculated, if it exists. In addition, the PDs of the symbols with $zz_y \neq 0$, are calculated and stored in a buffer $\mathcal{Q}$, of maximum size $\sqrt{|\mathcal{O}|}$. The symbol with the smallest PD in the buffer is the one to be visited first and removed from the buffer. If (zz_x^k, zz_y^k) are the zigzag coordinates of the k^{th} removed symbol from the buffer, to find the next one, the PD of the symbol with coordinates $(zz_x^k, zz_y^k + 1)$ is calculated (and put in the buffer). If $zz_y^k = 0$ the symbol with zigzag coordinates $(zz_x^k + 1, zz_y^k)$ is also computed. In the example of Fig. 2, if the SD partition requires examining the symbols with ordered positions from 12^{th} to 16^{th}, the symbols (1,3), (2,3) and (3,1) are first put in $\mathcal{Q}$. Since (1,3) is the one with the smallest PD in $\mathcal{Q}$, no new symbol is added because (1,4) does not exist. Then, the symbol with the second smallest PD in the buffer is visited, which is (3,1). After it is chosen, symbol (3,2) is added to $\mathcal{Q}$, and the process continues.

IV. Simulation Evaluations

This Section evaluates MultiSphere's performance via simulations[1]. Figure 3 shows MultiSphere's processing latency (in visited nodes)[2] and complexity (in PD calculations) for searching the SD tree and providing the ML solution. The examined SNRs range from 10 dB to 16 dB, corresponding to approximate symbol-error-rates (SERs) of $2 \cdot 10^{-1}$ and $3 \cdot 10^{-4}$ respectively. SD partitioning is performed based on the exact transmission channel. MultiSphere's latency and processing requirements are compared to those of the well-known ETH-SD [2] as well to those of the recently proposed Geosphere SD [3]. It is shown that, with 32 PEs, MultiSphere can achieve a latency reduction of more than an order of magnitude with overall complexity requirements similar to those of sequential SDs. Note that ETH and Geosphere SDs have the same latency requirements. MultiSphere is less complex than the ETH-SD but more complex than the highly optimized sequential Geosphere SD. In the high SNR regime, which is of higher interest due to corresponding SERs, MultiSphere with 32 PEs reaches a detection latency close to the one of linear detection methods (i.e., latency of n_t). Additionally, through SER simulations that are not shown due to space limitations, it has been verified that MultiSphere achieves the ML performance, in all shown cases.

To evaluate MultiSphere's efficiency, comparison have been performed using two SD partitioning methods. The same processing flow as MultiSphere has been applied since the authors are not aware of other nearly-embarrassingly parallel exact SDs. Parallel Method I uses a partitioning method similar

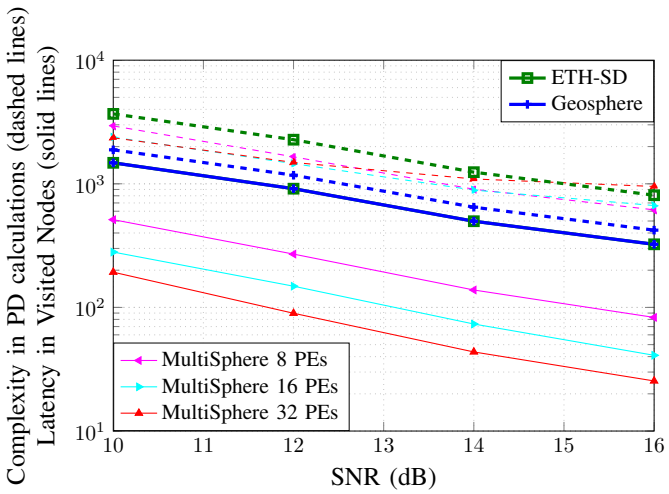

Fig. 3. MultiSphere's complexity and latency requirements vs. ETH SD and Geosphere SD.

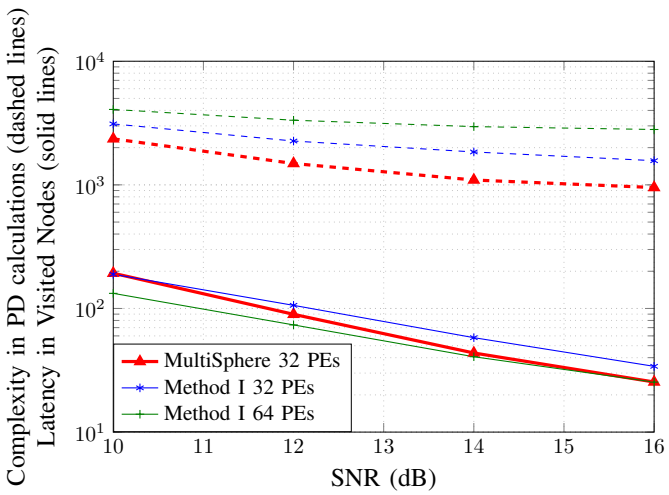

Fig. 4. MultiSphere's complexity and latency requirrments vs. Parallel Method I, with several numbers of PEs.

[1]We assume an uncoded, 16-QAM modulated 8×8 MIMO system using multi-carrier transmission with 64 subcarriers. Each sub-channel between a transmit-receive antenna pair is modeled as a 5 tap i.i.d. Rayleigh channel.

[2]We assume one-node-per-cycle architectures [2], therefore the latency can be measured either in visited nodes or in processing cycles. The latency for finding and distributing the minimum r^2 across parallel SDs is ignored.

to the one of fixed complexity SD [14] and of Yang et al. [12]. Nodes with the smallest PDs at the highest tree level are allocated to separate subtrees. If more PEs are available, nodes of lower tree levels are allocated to separate subtrees, starting from nodes whose parent has the smallest PD amongst its siblings, and continuing with children nodes of other parents in an ascending order of their PDs. Parallel Method II is a "K-Best-like" approach, according to which the nodes with the smallest PDs of the highest tree level are allocated to separate subtrees, similarly to Method I. If more PEs are available, Method II allocates equal number of children nodes from parents in separate subtrees. The children with the smallest PDs are allocated first. It is noted that when PD sorting is required during the tree traversal, if it is sequential, the complexity efficient method of Geosphere [3] SD is used for Methods I and II. Otherwise, exhaustive enumeration and sorting is performed. Figure 4 compares MultiSphere to Method I, and shows that MultiSphere consistently outperforms Method I both in terms of complexity and latency. Specifically in SNR regimes of higher interest while the MultiSphere achieves the latency of Method I with 64 PEs, by using half the PEs and an overall complexity which is 3 times smaller. Similar results, not shown here due to space limitations, hold for Method II.

For multi-carrier systems with a number of PEs smaller or equal to the number of subcarriers (SCs), parallelization can be traditionally performed by allocating one PE to each SC. Instead of this, MultiSphere can be used to parallelize the detection on each SC and then process SCs sequentially.

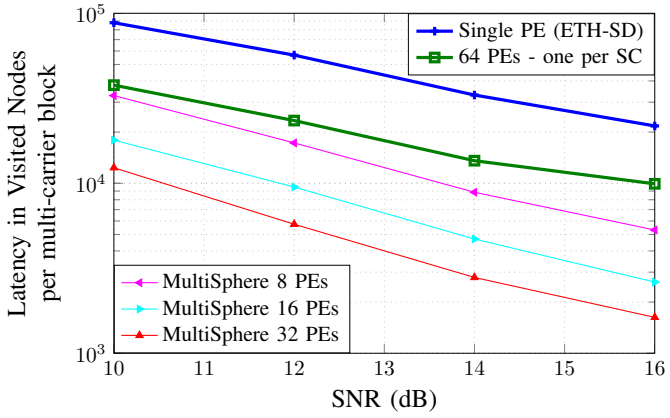

Fig. 5. Latency requirements for a multicarrier system with a PE per subcarrier vs. MultiSphere with sequential subcarrier processing.

Figure 5 shows the average latency per multi-carrier block (e.g., OFDM symbol) when using MultiSphere against a system which allocates a single PE for each SC (therefore uses 64 PEs). It can be seen that performing parallelization per subcarrier is rather inefficient. MultiSphere can reach smaller latencies by using less than 8 PEs, than by performing parallelization at a subcarrier level and using 64 PEs.

V. Conclusions

This work proposes MultiSphere, a method to consistently and massively parallelize large sphere decoders in a nearly-embarrassingly manner, while accounting for the transmission channel. It is shown that MultiSphere substantially reduces latency at a small complexity overhead. As a result MultiSphere finds applications in large MIMO systems and other non-orthogonal schemes and can also be extended to soft-input, soft-output sphere decoders with methods like [17].

Acknowledgment

The research leading to these results has been supported from the UK's Engineering and Physical Sciences Research Council (EPSRC Grant EP/M029441/1). The Authors would like to also thank the members of University of Surrey 5GIC (http://www.surrey.ac.uk/5GIC) for their support.

References

[1] E. Viterbo and J. Boutros, "A universal lattice code decoder for fading channels," *IEEE Trans. Inf. Theory*, vol. 45, no. 5, pp. 1639–1642, 1999.

[2] A. Burg, M. Borgmann, M. Wenk, M. Zellweger, W. Fichtner, and H. Bolcskei, "VLSI implementation of MIMO detection using the sphere decoding algorithm," *IEEE J. of Solid-State Circ.*, vol. 40, no. 7, pp. 1566–1577, 2005.

[3] K. Nikitopoulos, J. Zhou, B. Congdon, and K. Jamieson, "Geosphere: Consistently turning MIMO capacity into throughput," in *Proc. of the 2014 ACM SIGCOMM*, 2014, pp. 631–642.

[4] T. Skotnicki, J. A. Hutchby, T.-J. King, H. Wong, and F. Boeuf, "The end of CMOS scaling: toward the introduction of new materials and structural changes to improve MOSFET performance," *IEEE Circuits and Devices Magazine*, vol. 21, no. 1, pp. 16–26, 2005.

[5] G. Fettweis, "5G–what will it be: the tactile internet," in *Proc. of IEEE ICC*, 2013.

[6] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[7] C. Hess *et al.*, "Reduced-complexity MIMO detector with close-to-ML error rate performance," in *Proc. of ACM Great Lakes VLSI Symp.*, 2008.

[8] Z. Guo and P. Nilsson, "Algorithm and implementation of the K-best sphere decoding for MIMO detection," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 3, pp. 491–503, 2006.

[9] M. S. Khairy, C. Mehlfhrer, and M. Rupp, "Boosting sphere decoding speed through graphic processing units," in *Proc. of European Wireless Conference (EW)*, April 2010, pp. 99–104.

[10] M. Wu, S. Gupta, Y. Sun, and J. R. Cavallaro, "A GPU implementation of a real-time MIMO detector," in *Proc. of IEEE Workshop on Signal Processing Systems*, Oct 2009, pp. 303–308.

[11] C. M. Józsa, G. Kolumbán, A. M. Vidal, F.-J. Martínez-Zaldívar, and A. González, "New parallel sphere detector algorithm providing high-throughput for optimal MIMO detection," *Procedia Computer Science*, vol. 18, pp. 2432 – 2435, 2013.

[12] C.-H. Yang and D. Marković, "A multi-core sphere decoder VLSI architecture for MIMO communications," in *Proc. of IEEE GLOBECOM*. IEEE, 2008, pp. 1–6.

[13] C. H. Yang and D. Marković, "A 2.89mW 50GOPS 16x16 16-core MIMO sphere decoder in 90nm CMOS," in *Proc. of ESSCIRC*, Sept 2009, pp. 344–347.

[14] L. G. Barbero and J. S. Thompson, "Fixing the complexity of the sphere decoder for MIMO detection," *IEEE Transactions on Wireless Communications*, vol. 7, no. 6, pp. 2131–2142, 2008.

[15] K. Nikitopoulos, A. Karachalios, and D. Reisis, "Exact max-log MAP soft-output sphere decoding via approximate Schnorr–Euchner enumeration," *IEEE Transactions on Vehicular Technology*, vol. 64, no. 6, pp. 2749–2753, 2015.

[16] C. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Math. Prog.*, vol. 66, no. 2, pp. 181–191, 1994.

[17] K. Nikitopoulos *et al.*, "Complexity-efficient enumeration techniques for soft-input, soft-output sphere decoding," *IEEE Comms. L.*, vol. 14, no. 4, pp. 312–314, 2010.