Parallel computation of entries of A^{-1}

François-Henry Rouet
ENSEEIHT-IRIT, Université de Toulouse, France

Joint work with: P. Amestoy, I. Duff, J.-Y. L'Excellent & B. Uçar

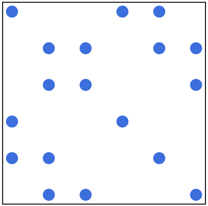
SIAM Workshop on Combinatorial Scientific Computing, May 19-21 2011

Computation of entries of A^{-1}

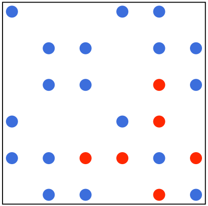
Problem

- Given a large sparse matrix A, compute a set of entries of A^{-1}.
- Many applications: linear least-squares, quantum-scale device simulation, short-circuit currents, astrophysics...
- Typical case: computation of the whole diagonal of A^{-1}.

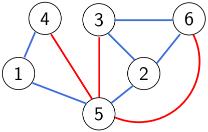
We rely on the pattern of A and its factors L and U such that $A = LU$.



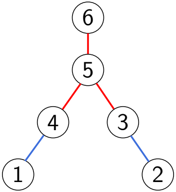
Pattern of A.



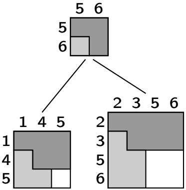
Pattern of $L + U$.



Graph of $L + U$.



Elimination tree.



Assembly tree.

Computing an entry in A^{-1}

The (i, j) entry of A^{-1} is computed as $a_{i,j}^{-1} = (A^{-1}e_j)_i$.
Using the LU factors,

$$\begin{cases} y = L^{-1}e_j \\ a_{i,j}^{-1} = (U^{-1}y)_i \end{cases}$$

Computing an entry in A^{-1}

The (i, j) entry of A^{-1} is computed as $a^{-1}_{i,j} = (A^{-1}e_j)_i$.
Using the LU factors,

$$\begin{cases} y = L^{-1}e_j & \Rightarrow \text{sparse right-hand side.} \\ a^{-1}_{i,j} = (U^{-1}y)_i & \Rightarrow \text{only one component needed.} \end{cases}$$

Computing an entry in A^{-1}

The (i,j) entry of A^{-1} is computed as $a^{-1}_{i,j} = (A^{-1}e_j)_i$.
Using the LU factors,

$$\begin{cases} y = L^{-1}e_j & \Rightarrow \text{sparse right-hand side.} \\ a^{-1}_{i,j} = (U^{-1}y)_i & \Rightarrow \text{only one component needed.} \end{cases}$$

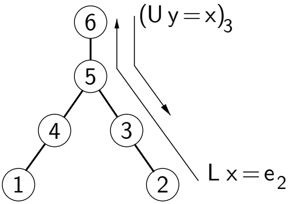
Sparsity of both the RHS and the solution is exploited to reduce the traversal of the tree:

For each requested entry $a^{-1}_{i,j}$,

(1) visit the nodes of the elimination tree from node j to the root: at each node access necessary parts of L,

(2) visit the nodes from the root to node i; this time access necessary parts of U.

$\Rightarrow$ "pruned tree"



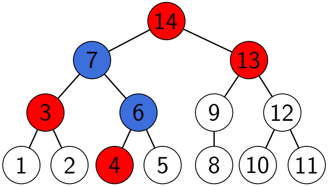
Computation of $a^{-1}_{3,2}$.

Computing a set of entries

In reality

We wish to compute a set R of requested entries. Usually $|R|$ is large and one cannot hold all the solution vectors in memory, even with a storage scheme that exploits sparsity. We assume that we process nb solution vectors at a time.

The way the requested entries are partitioned has a strong influence on the number of accesses to the nodes:



$[R = \{3, 4, 13, 14\},\ nb = 3]$

	Partition	Accesses
Π'	$R_1 = \{3, 13, 14\}$ $R_2 = \{4\}$	$R_1 : 3, 7, 13, 14$ $R_2 : 4, 6, 7, 14$
Π''	$R_1 = \{3, 4, 14\}$ $R_2 = \{13\}$	$R_1 : 3, 4, 6, 7, 14$ $R_2 : 13, 14$

Tree-partitioning problem in out-of-core

Tree-Partitioning problem (OOC version)

Given a set R of nodes of a node-weighted tree and a blocksize nb, find a partition $\Pi(R) = \{R_1, R_2, \ldots\}$ such that $\forall R_k \in \Pi, |R_k| \leq nb$, and has minimum cost

$$\mathrm{Cost}(\Pi) = \sum_{R_k \in \Pi} \mathrm{Cost}(R_k) \quad \text{where} \quad \mathrm{Cost}(R_k) = \sum_{i \in P(R_k)} w(i)$$

- We showed that it is NP-complete.
- There is a non-trivial lower bound.
- The case $nb = 2$ is special and can be solved in polynomial time.
- A simple algorithm, postorder, gives an approximation guarantee.
- We have a heuristic which gives extremely good results.
- We have hypergraph models that address the most general cases.

P. Amestoy, I. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. R. and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. Submitted to SIAM journal on Scientific Computing.

Parallel issues

Computing blocks in parallel ?

Computing (many) blocks is embarassingly parallel: one would like to compute all the blocks in parallel, but:

- In a distributed memory environment, this is not feasible without replicating the factors.
- In a shared-memory environment, this is feasible but might lead to poor performance (memory demanding).

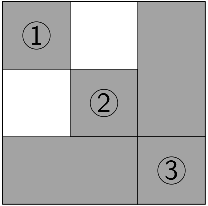
Computational setting

Blocks are processed one by one:

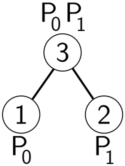
- Sparsity is exploited between the blocks, i.e. the tree is pruned for each block.
- Sparsity is not exploited within the blocks, to benefit from dense kernels (`BLAS`).

Parallel issues - cont

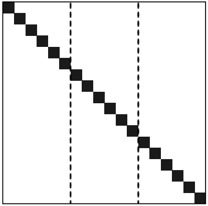
Problem: any permutation aimed at reducing flops provides poor speed-ups.
Archetypal example: whole diagonal of A^{-1}, with $nb = N/3$.



[Simple matrix…



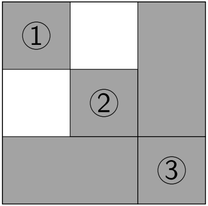
…and its tree.]



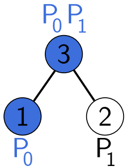
[Right-hand sides]

Parallel issues - cont

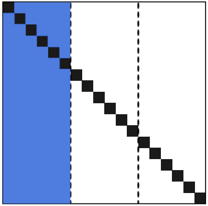
Problem: any permutation aimed at reducing flops provides poor speed-ups.
Archetypal example: whole diagonal of A^{-1}, with $nb = N/3$.



[Simple matrix. . .

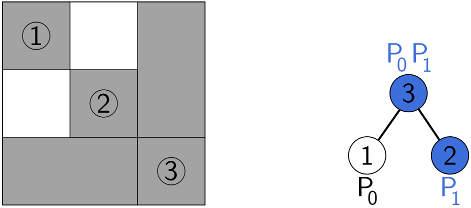


. . . and its tree.]

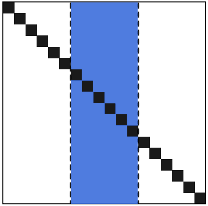

[Right-hand sides]

1^{st} block: traverses nodes 1 and 3, only P_0 active at the bottom of the tree.

Problem: any permutation aimed at reducing flops provides poor speed-ups.
Archetypal example: whole diagonal of A^{-1}, with $nb = N/3$.



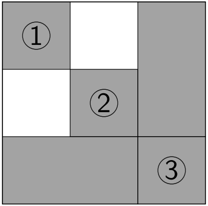
[Simple matrix. . . . . . and its tree.]


[Right-hand sides]

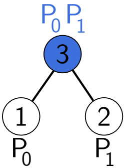
1^{st} block: traverses nodes 1 and 3, only P_0 active at the bottom of the tree.

2^{nd} block: traverses nodes 2 and 3, only P_1 active at the bottom of the tree.

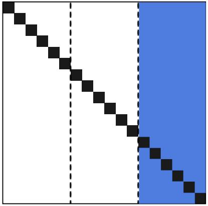
Problem: any permutation aimed at reducing flops provides poor speed-ups.
Archetypal example: whole diagonal of A^{-1}, with $nb = N/3$.



[Simple matrix...



...and its tree.]



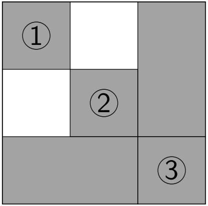
[Right-hand sides]

1^{st} block: traverses nodes 1 and 3, only P_0 active at the bottom of the tree.

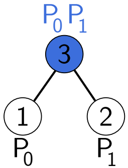
2^{nd} block: traverses nodes 2 and 3, only P_1 active at the bottom of the tree.

3^{rd} block: traverses node 3.

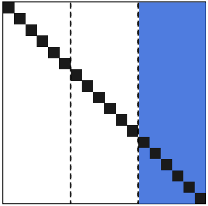
Problem: any permutation aimed at reducing flops provides poor speed-ups.
Archetypal example: whole diagonal of A^{-1}, with $nb = N/3$.



[Simple matrix...



...and its tree.]



[Right-hand sides]

1^{st} block: traverses nodes 1 and 3, only P_0 active at the bottom of the tree.

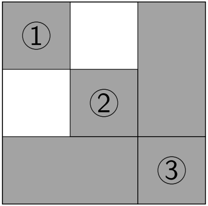
2^{nd} block: traverses nodes 2 and 3, only P_1 active at the bottom of the tree.

3^{rd} block: traverses node 3.

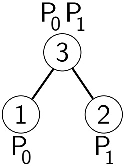
Few active procs at the bottom of the tree

$\Rightarrow$ poor speed-up.

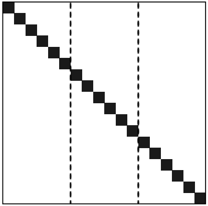
An attempt (Slavova): interleave the requested entries over the processors.



[Simple matrix. . .

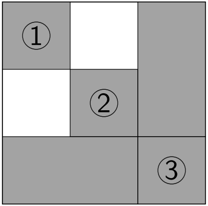


. . . and its tree.]

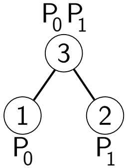


[Right-hand sides]

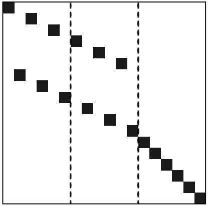
An attempt (Slavova): interleave the requested entries over the processors.



[Simple matrix...

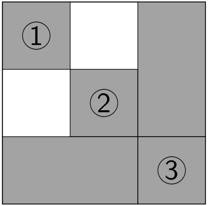


...and its tree.]

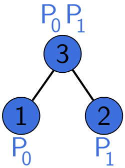


[Right-hand sides]

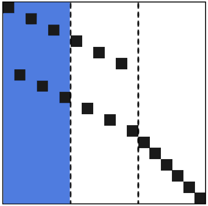
An attempt (Slavova): interleave the requested entries over the processors.



[Simple matrix. . .



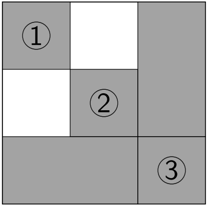
. . . and its tree.]



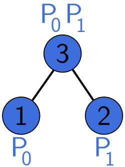
[Right-hand sides]

1^{st} block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.

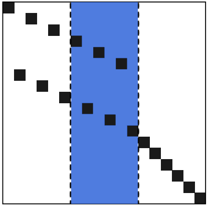
An attempt (Slavova): interleave the requested entries over the processors.



[Simple matrix...



...and its tree.]



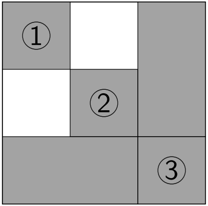
[Right-hand sides]

1st block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.

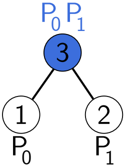
2nd block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.

Parallel issues - cont

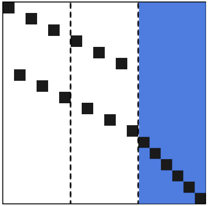
An attempt (Slavova): interleave the requested entries over the processors.



[Simple matrix. . .



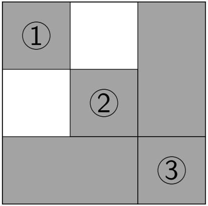
. . . and its tree.]



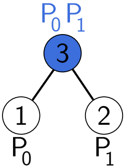
[Right-hand sides]

- **1^{st} block:** traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
- **2^{nd} block:** traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
- **3^{rd} block:** traverses node 3.

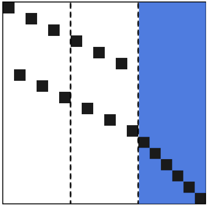
An attempt (Slavova): interleave the requested entries over the processors.



[Simple matrix. . .



. . . and its tree.]



[Right-hand sides]

1[st] block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.

2[nd] block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.

3[rd] block: traverses node 3.

More active procs but more flops !

$\Rightarrow$ no speed-up.

Exploiting sparsity within the blocks

Core idea

- Interleaving tends to cancel the benefits of a good permutation, since it groups together entries that are distant in the tree.

Exploiting sparsity within the blocks

Core idea

- Interleaving tends to cancel the benefits of a good permutation, since it groups together entries that are distant in the tree.
- To prevent this, one can choose to exploit sparsity within a block.
  Key: at each node, operations will be performed on the necessary columns only (instead of the whole block)
  - Right-hand sides are still processed by blocks of nb
  - Dense computations are performed on subblocks of size called nb_{sparse}

Exploiting sparsity within the blocks

Core idea

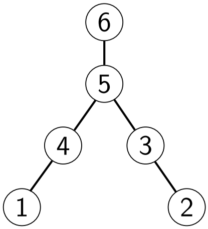
- Interleaving tends to cancel the benefits of a good permutation, since it groups together entries that are distant in the tree.
- To prevent this, one can choose to exploit sparsity within a block.
  Key: at each node, operations will be performed on the necessary columns only (instead of the whole block)
  - Right-hand sides are still processed by blocks of nb
  - Dense computations are performed on subblocks of size called nb_{sparse}
- Each node of the tree is provided with the subscripts of the columns to process. We do not want to manage a list; to ensure efficiency, we rely on the interval that bounds the list of necessary columns.

Exploiting sparsity within the blocks - cont

Core idea - cont

- Intervals are computed in a two-step traversal of the tree:

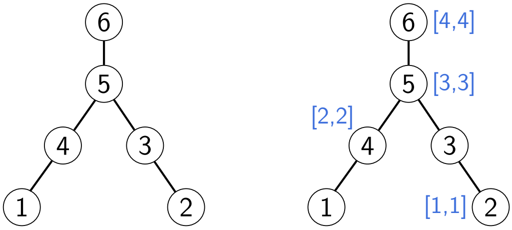
Example: the block of right-hand sides is equal to $[e_2, e_4, e_5, e_6]$.



Core idea - cont

- Intervals are computed in a two-step traversal of the tree:
  1. Initialization: at each node, initialize with the target entries (columns) appearing at the node.

Example: the block of right-hand sides is equal to $[e_2, e_4, e_5, e_6]$.

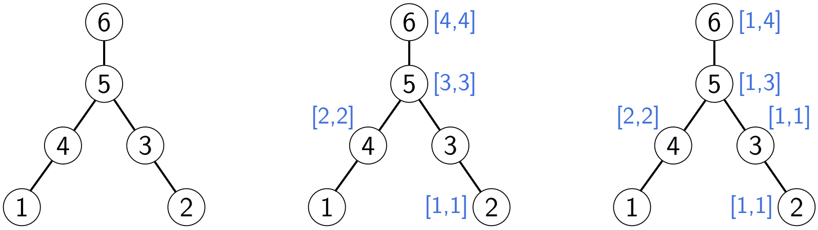


Exploiting sparsity within the blocks - cont

Core idea - cont

- Intervals are computed in a two-step traversal of the tree:
    1. Initialization: at each node, initialize with the target entries (columns) appearing at the node.
    2. Propagation: at each node, add the union of the intervals of its children.

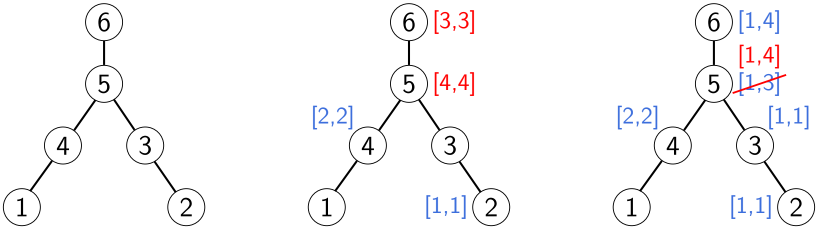
Example: the block of right-hand sides is equal to $[e_2, e_4, e_5, e_6]$.



Core idea - cont

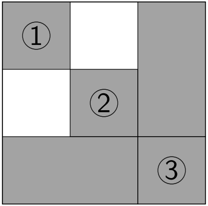
- Intervals are computed in a two-step traversal of the tree:
  1. Initialization: at each node, initialize with the target entries (columns) appearing at the node.
  2. Propagation: at each node, add the union of the intervals of its children.
- By postordering each block, this interval is reduced.

Example: with a non-postordered block $[e_2, e_4, e_6, e_5]$

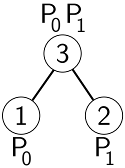


Illustration

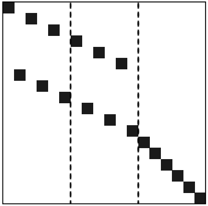
Back to the first example: $nb = N/3$, we use interleaving and blocks are postordered; when computing a block, compute for each node the interval to process.



[Simple matrix. . .

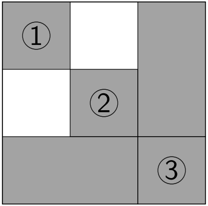


. . . and its tree.]

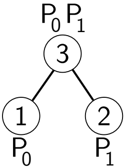


[Right-hand sides]

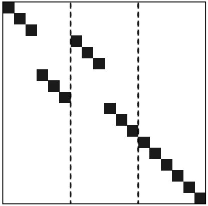
Back to the first example: $nb = N/3$, we use interleaving and blocks are postordered; when computing a block, compute for each node the interval to process.



[Simple matrix. . .



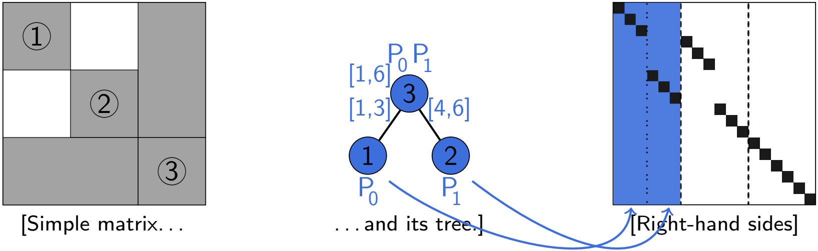
. . . and its tree.]



[Right-hand sides]

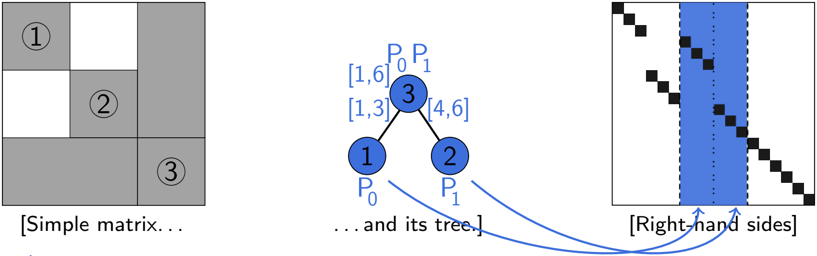
Illustration

Back to the first example: $nb = N/3$, we use interleaving and blocks are postordered; when computing a block, compute for each node the interval to process.



1^{st} block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
At node 1, flops are performed on $N/6$ vectors only. Same at node 2.

Back to the first example: $nb = N/3$, we use interleaving and blocks are postordered; when computing a block, compute for each node the interval to process.



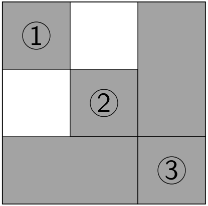
[Simple matrix… …and its tree.] [Right-hand sides]

1[st] block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
At node 1, flops are performed on $N/6$ vectors only. Same at node 2.

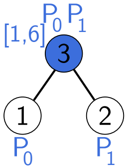
2[nd] block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
At node 1, flops are performed on $N/6$ vectors only. Same at node 2.

Illustration

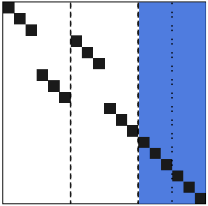
Back to the first example: $nb = N/3$, we use interleaving and blocks are postordered; when computing a block, compute for each node the interval to process.



[Simple matrix...



...and its tree.]

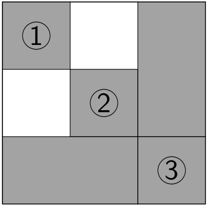


[Right-hand sides]

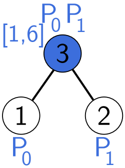
1st block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
At node 1, flops are performed on $N/6$ vectors only. Same at node 2.

2nd block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
At node 1, flops are performed on $N/6$ vectors only. Same at node 2.

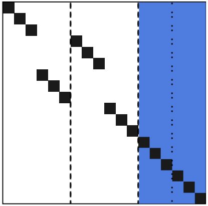
Back to the first example: $nb = N/3$, we use interleaving and blocks are postordered; when computing a block, compute for each node the interval to process.



[Simple matrix. . .



. . . and its tree.]



[Right-hand sides]

1st block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
At node 1, flops are performed on $N/6$ vectors only. Same at node 2.

2nd block: traverses all the nodes, P_0 and P_1 active at the bottom of the tree.
At node 1, flops are performed on $N/6$ vectors only. Same at node 2.

All procs are active, and flops have not increased !

$\Rightarrow$ good speed-up.

Experiments

The different strategies are compared. The average number of active processors at leaf nodes of the pruned tree is used as a measure of tree parallelism.

Computation of 10% diagonal entries, 11-point discretization of a $200 \times 200 \times 20$ grid, blocks of size 512:

Procs	Strategy		Time (seconds)	Operation (TFlops)	Active procs
1	-		1667	16.2	1
4	IL off	nb_{sparse} off	1366	16.2	1.10
	IL on	nb_{sparse} off	2028	45.4	3.92
		nb_{sparse} on	659	15.2	
8	IL off	nb_{sparse} off	1241	13.3	1
	IL on	nb_{sparse} off	1508	61.0	7.76
		nb_{sparse} on	418	12.4	

Influence of the block size on the same problem:

Procs	Strategy		Block size			
			64	128	512	1024
1 proc	-		1518	1432	1667	2002
8 procs	IL on	nb_{sparse} on	555	466	418	379

Conclusion

Exploiting sparsity within a block. . .

. . . can be done without sacrificing efficiency (BLAS kernels optimally used).

. . . increases parallelism when combined with interleaving.

. . . is interesting even in the sequential case (it reduces flops).

. . . gives some leeway for the backward phase (off-diagonal case): each block can be reordered following a permutation that will have a good effect for backward targets.

Conclusion - cont

Further work

- Still some effort to make to reach the scalability of the dense solve.
- Several improvements upon interleaving: management of type 2 nodes, management of sequential subtrees...
- Minimize nb_{sparse}-sized intervals in the general case, i.e. find a good permutation within each block (postorder works fine for diagonal entries...).

Next release of MUMPS

- Compressed solution space when exploiting sparse right-hand sides.
- Use of sparsity within blocks of sparse right-hand sides.

Thank you for your attention!

Any questions?