Block Wiedemann algorithm on multicore architectures
Bastien Vialla

To cite this version:

HAL Id: lirmm-01372535
https://hal-lirmm.ccsd.cnrs.fr/lirmm-01372535
Submitted on 27 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Notations

- \( K \) a commutative field.
- \( M_{n\times m}(K) \), ring of matrices of size \( n \times m \).
- We denote by \( \gamma \) the number of non zeros in a sparse matrix.
- NUMA : Non Uniform Memory Access

Motivations

Solving a linear system with large sparse matrices is a computational kernel used in a wide range of applications. The block version of Wiedemann’s algorithm proposed in [1] take advantage of the sparsity to achieve better performance.

Objectives

An efficient implementation of block Wiedemann algorithm on NUMA multicores architectures.

Contribution

- We efficiently incorporate the sparse block into the first step of BW algorithm.
- We provide an efficient implementation for NUMA multicores using tbb/MPI that provides excellent scaling.

Block Wiedemann algorithm

Let \( A \in M_{n \times n}(K) \), \( U, V \in M_{n \times k}(K) \) random matrices with \( k \leq n \).

Block Wiedemann algorithm follows three steps:

1. Compute the first \( O(\frac{n^2}{k}) \) elements of \( S = [U^T A V]_{\in \mathbb{N}} \).
2. Find the minimal matrix polynomial generator of the sequence \( S \).
3. Compute the solution using the polynomial found in step 2.

The cost of the first step is dominant, therefore its parallelization is crucial.

Sparse blocks

We generalize sparse block from [2] in block Wiedemann algorithm. We permute non zeros elements to have a cache efficient version.

The block Wiedemann Algorithm follows three steps:

- Compute the first \( O(\frac{n^2}{k}) \) elements of \( S = [U^T A V]_{\in \mathbb{N}} \).
- Find the minimal matrix polynomial generator of the sequence \( S \).
- Compute the solution using the polynomial found in step 2.

The block size does not impact the time with the use of sparse blocks, see figure 1.

tbb implementation performs badly on NUMA architectures, see table 1.

We design an Hybrid MPI/tbb implementation that allows good speed-up on NUMA, see table 1.

Sparse blocks perform always better, see table 1.

Complexities of the first step of block Wiedemann algorithm:

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel k cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense blocks</td>
<td>( O(n^\gamma + n^k) )</td>
</tr>
<tr>
<td>Sparse blocks</td>
<td>( O(\frac{n^\gamma + n^k}{k}) )</td>
</tr>
</tbody>
</table>

where \( s = \lfloor n/k \rfloor \), \( \delta_1, \ldots, \delta_{s+1} \in K \) chosen at random.

Experiments

- We use LinBox [3] for dense blocks code, and tbb for parallelization. We use an NUMA with four Intel XEON E4620 with 8 cores at 2.2Ghz and 384GB of RAM.
- The block size does not impact the time with the use of sparse blocks, see figure 1.
- tbb implementation performs badly on NUMA architectures, see table 1.
- We design an Hybrid MPI/tbb implementation that allows good speed-up on NUMA, see table 1.

Figure 1: Sparse blocks size influence in comparison with dense blocks. Computations made with the matrix EX5 with 1 core.

<table>
<thead>
<tr>
<th>Dense blocks (LinBox)</th>
<th>Sparse blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>tbb</td>
<td>tbb</td>
</tr>
<tr>
<td>time</td>
<td>speed-up</td>
</tr>
<tr>
<td>1 cpu : 1 core</td>
<td>2205</td>
</tr>
<tr>
<td>1 cpu : 8 cores</td>
<td>540</td>
</tr>
<tr>
<td>2 cpus : 16 cores</td>
<td>843</td>
</tr>
<tr>
<td>3 cpus : 24 cores</td>
<td>708</td>
</tr>
<tr>
<td>4 cpus : 32 cores</td>
<td>960</td>
</tr>
</tbody>
</table>

Table 1: Timings, in seconds, and speed-up of tbb and MPI/tbb implementations. We use the matrix rand100k.

Additional Informations

Matrices characteristics:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Non zeros</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand100k</td>
<td>100k x 100k</td>
<td>1.5M</td>
<td>randomly generated</td>
</tr>
<tr>
<td>EX5</td>
<td>64x64 x 64x64</td>
<td>20G</td>
<td>symmetric powers of graphs</td>
</tr>
</tbody>
</table>

References