Scaling the Memory Wall for Sparse Iterative Solvers

Jonas Thies
Delft High Performance Computing Center

Christie Alappatt, Gerhard Wellein & Georg Hager
University of Erlangen-Nuremberg
A “Bread & Butter” Simulation

A “Bread & Butter” Simulation

1m resolution (4M grid cells)

A “Bread & Butter” Simulation

1m resolution (4M grid cells)

Actuator-Line Model

A “Bread & Butter” Simulation

1m resolution
(4M grid cells)

Actuator-Line Model

Implicit Time-Stepping

A “Bread & Butter” Simulation

1m resolution (4M grid cells)

Actuator-Line Model

Implicit Time-Stepping

Two Sparse Systems per Time-Step

GMRES for the Momentum Equations

Build orthogonal basis of a Krylov subspace

\[ \mathcal{K}_m(A, r_0) = \text{span}\{q_1, q_2, \ldots, q_m\} \]

Using the Arnoldi process

\[ AQ_k = Q_{k+1}H_k. \]
GMRES for the Momentum Equations

Build orthogonal basis of a Krylov subspace

\[ \mathcal{K}_m(A, r_0) = \text{span}\{q_1, q_2, \ldots, q_m\} \]

Using the Arnoldi process

\[ AQ_k = Q_{k+1} H_k. \]

One vector requires 64MB of RAM
GMRES for the Momentum Equations

Build orthogonal basis of a Krylov subspace

\[ \mathcal{K}_m(A, r_0) = \text{span}\{q_1, q_2, \ldots, q_m\} \]

Using the Arnoldi process

\[ AQ_k = Q_{k+1}H_k. \]

One vector requires 64MB of RAM

Needs only 20-40 Iterations
GMRES for the Momentum Equations

Build orthogonal basis of a Krylov subspace

\[ \mathcal{K}_m(A, r_0) = \text{span}\{q_1, q_2, \ldots, q_m\} \]

Using the Arnoldi process

\[ AQ_k = Q_{k+1}H_k. \]

One vector requires 64MB of RAM

Needs only 20-40 iterations

Runtime breakdown on 48-core Xeon (1536 Flops/cycle)
GMRES for the Momentum Equations

Build orthogonal basis of a Krylov subspace

\[ K_m(A, r_0) = \text{span}\{q_1, q_2, \ldots, q_m\} \]

Using the Arnoldi process

\[ AQ_k = Q_{k+1} H_k. \]

One vector requires 64MB of RAM

Needs only 20-40 Iterations

What is “Communication”, And how can it be “avoided”

Runtime breakdown on 48-core Xeon (1536 Flops/cycle)

- dot
- SpMV
- axpby
“Communication Avoiding” Krylov

- Partially use polynomial with constant coefficients
- Fewer orthogonalization operations
- More compute-intensive ‘tall&skinny’ operations
- Bundle reductions
- **Glue SpMV’s together (Matrix-Power Kernel):**

\[ y^k = \sum_{k=1}^{s} \alpha_k A^k x \]
“Communication Avoiding” Krylov

- Use polynomial with constant coefficients
- Fewer orthogonalization operations
- More compute-intensive ‘tall&skinny’ operations
- Bundle reductions
- Glue SpMV’s together (Matrix-Power Kernel):

\[ y^k = \sum_{k=1}^{s} \alpha_k A^k x \]
“Communication Avoiding” Krylov

- Synchronize polynomial with constant coefficients
- Fewer orthogonalization operations
- More compute-intensive ‘tall-skinny’ operations
- Bundle reductions
- Glue SpMVs together (Matrix-Power Kernel):

\[ y^k = \sum_{k=1}^{s} \alpha_k A^k x \]
“Communication Avoiding” Krylov

- Use transpose polynomial with constant coefficients
- Fewer orthogonalization operations
- More compute-intensive ‘tall&skinny’ operations
- Bundle reductions
- Global SVD

Glue SpMV’s together (Matrix-Power Kernel):

\[ y^k = \sum_{k=1}^{s} \alpha_k A^k x \]
“Communication Avoiding” Krylov

- Use polynomial with constant coefficients
- Fewer orthogonalization operations
- More compute-intensive ‘tall&skinny’ operations
- Bundle reductions
- Glue SpMV’s together (Matrix-By-Vector Kernel):
  - Polynomial preconditioners
  - s-step methods

Synchronization Points

Communication Volume
Simple Example:
Neumann Polynomial Preconditioner
Simple Example: Neumann Polynomial Preconditioner

\[ \frac{1}{x} \approx \sum_{k=1}^{s} (1 - x)^k \]
Simple Example: Neumann Polynomial Preconditioner

Neumann series

\[ \frac{1}{x} \approx \sum_{k=1}^{s} (1 - x)^k \]

Jacobi splitting

\[ A = D - (L + U) \]
Simple Example: Neumann Polynomial Preconditioner

Neumann series
\[
\frac{1}{x} \approx \sum_{k=1}^{s} (1 - x)^k
\]

Jacobi splitting
\[
A = D - (L + U)
\]

Degree-s preconditioner
\[
(D^{-1} A)^{-1} v \approx \left[ D^{-1} (L + U) \right]^{s} D^{-1} v
\]
Matrix Power Kernel

"Back to back" SpMVs

RACE MPK

\[ x \]
Matrix Power Kernel

“Back to back” SpMVs

Ax \quad = \quad x

RACE MPK

\begin{align*}
\end{align*}
Matrix Power Kernel

“Back to back” SpMVs

\[ \text{Ax} \]

RACE MPK

\[ = \]
Matrix Power Kernel

"Back to back" SpMVs

\[ A^{2x} = \]

RACE MPK

\[ = \]
Matrix Power Kernel

“Back to back” SpMVs

\[ A^2x \]

RACE MPK
Matrix Power Kernel

"Back to back" SpMVs

\[ A^2 x \]

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

"Back to back" SpMVs

Matrix accessed 3 times from memory

RACE MPK

Matrix accessed 3 times from memory

$X$
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

\[ x \]

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

Matrix accessed 3 times from memory X
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

Matrix accessed 3 times from memory
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

x
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

x
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

x
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

"Back to back" SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

\[ \text{RACE MPK} \]

\[ \text{Matrix accessed 3 times from memory} \times \]
Matrix Power Kernel

"Back to back" SpMVs

Matrix accessed 3 times from memory

X

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

RACE MPK

Matrix accessed 3 times from memory

$x^X$
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

=
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK
Matrix Power Kernel

“Back to back” SpMVs

RACE MPK

Matrix accessed 3 times from memory

\[ x \]
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

= $x^n$
Matrix Power Kernel

“Back to back” SpMVs

Matrix accessed 3 times from memory

RACE MPK

Matrix accessed 1 time from memory
Matrix Power Kernel

"Back to back" SpMVs

Matrix accessed 3 times from memory

RACE MPK

Matrix accessed 1 time from memory


Wavefront passing through matrix.
- More complicated than the Regular example seen here!
Sparse Matrix Graph

Example of 2D-7 Point stencil
Sparse Matrix → Graph

Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
RACE: Recursive Algebraic Coloring

Example of 2D-7 Point stencil
RACE MPK Performance

256 GB RAM (146 GB/s)
RACE MPK Performance

- 38 cores
- 57MB L3 Cache (420 GB/s)
- 256 GB RAM (146 GB/s)

Graph showing performance comparison between Baseline and RACE for Power ($p_m$) from 1 to 10.
RACE MPK Performance

38 cores

57MB L3 Cache (420 GB/s)

256 GB RAM (146 GB/s)

Baseline

RACE

Power ($p_m$)
RACE MPK Performance

- Intel Xeon Gold Inside
- 38 cores
- 57MB L3 Cache (420 GB/s)
- 256 GB RAM (146 GB/s)

Graph showing performance comparison between Baseline and RACE.

Power ($p_m$) vs Performance.
RACE MPK Performance

- **38 cores**
- **57MB L3 Cache (420 GB/s)**
- **256 GB RAM (146 GB/s)**

Graph showing performance comparison between Baseline and RACE with respect to power ($p_m$).
RACE MPK Performance

- 38 cores
- 57MB L3 Cache (420 GB/s)
- 256 GB RAM (146 GB/s)

Graph:
- Baseline
- RACE

Power ($p_m$)
RACE MPK Performance

- Intel Xeon Gold inside
- 38 cores
- 57MB L3 Cache (420 GB/s)
- 256 GB RAM (146 GB/s)

Graph showing performance comparison between Baseline and RACE.

Power (\(p_m\))
RACE MPK Performance

- 38 cores
- 57MB L3 Cache (420 GB/s)
- 256 GB RAM (146 GB/s)

Graph showing performance comparison between Baseline and RACE over different Power levels ($p_m$).
RACE MPK Performance

256 GB RAM (146 GB/s)
RACE MPK Performance

- 64 cores
- 256 MB L3 Cache (2600 GB/s)
- 256 GB RAM (146 GB/s)

Graph showing performance over power (\(p_m\)).
Example: Nalu-Wind Model

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMV</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD EPYC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Nalu-Wind Model

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMV</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>43</td>
<td>43</td>
<td>3.72</td>
<td>3.83</td>
</tr>
</tbody>
</table>
Example: Nalu-Wind Model

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMV</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>43</td>
<td>43</td>
<td>3.72</td>
<td>3.83</td>
</tr>
<tr>
<td>Jacobi</td>
<td>24</td>
<td>24</td>
<td>1.73</td>
<td>1.83</td>
</tr>
</tbody>
</table>
Example: Nalu-Wind Model

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMV</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>43</td>
<td>43</td>
<td>3.72</td>
<td>3.83</td>
</tr>
<tr>
<td>Jacobi</td>
<td>24</td>
<td>24</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>GS</td>
<td>13</td>
<td>26</td>
<td>1.73</td>
<td>1.83</td>
</tr>
</tbody>
</table>
## Example: Nalu-Wind Model

![AMD EPYC Processor]

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMV</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>43</td>
<td>43</td>
<td>3.72</td>
<td>3.83</td>
</tr>
<tr>
<td>Jacobi</td>
<td>24</td>
<td>24</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>GS</td>
<td>13</td>
<td>26</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>Polv(5)</td>
<td>6</td>
<td>30</td>
<td>1.76</td>
<td>2.24</td>
</tr>
</tbody>
</table>
### Example: Nalu-Wind Model

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMVs</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>43</td>
<td>43</td>
<td>3.72</td>
<td>3.83</td>
</tr>
<tr>
<td>Jacobi</td>
<td>24</td>
<td>24</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>GS</td>
<td>13</td>
<td>26</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>Poly(5)</td>
<td>6</td>
<td>30</td>
<td>1.76</td>
<td>2.24</td>
</tr>
<tr>
<td>RACE + Poly(5)</td>
<td>6</td>
<td>30</td>
<td>1.16</td>
<td>1.64</td>
</tr>
</tbody>
</table>
Example: Nalu-Wind Model

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMVs</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>43</td>
<td>43</td>
<td>3.72</td>
<td>3.83</td>
</tr>
<tr>
<td>Jacobi</td>
<td>24</td>
<td>24</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>GS</td>
<td>13</td>
<td>26</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>Poly(5)</td>
<td>6</td>
<td>30</td>
<td>1.76</td>
<td>2.24</td>
</tr>
<tr>
<td>RACE + Poly(5)</td>
<td>6</td>
<td>30</td>
<td>1.16</td>
<td>1.64</td>
</tr>
<tr>
<td>Jacobi(5)</td>
<td>5</td>
<td>25</td>
<td>1.29</td>
<td>1.40</td>
</tr>
</tbody>
</table>
Example: Nalu-Wind Model

<table>
<thead>
<tr>
<th>Preconditioner</th>
<th>Iter.</th>
<th>#effective SpMVs</th>
<th>Solve time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>43</td>
<td>43</td>
<td>3.72</td>
<td>3.83</td>
</tr>
<tr>
<td>Jacobi</td>
<td>24</td>
<td>24</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>GS</td>
<td>13</td>
<td>26</td>
<td>1.73</td>
<td>1.83</td>
</tr>
<tr>
<td>Poly(5)</td>
<td>6</td>
<td>30</td>
<td>1.76</td>
<td>2.24</td>
</tr>
<tr>
<td>RACE + Poly(5)</td>
<td>6</td>
<td>30</td>
<td>1.16</td>
<td>1.64</td>
</tr>
<tr>
<td>Jacobi(5)</td>
<td>5</td>
<td>25</td>
<td>1.29</td>
<td>1.40</td>
</tr>
<tr>
<td>RACE + Jacobi(5)</td>
<td>5</td>
<td>25</td>
<td>0.60</td>
<td>0.70</td>
</tr>
</tbody>
</table>
What else can we do with RACE?
What else can we do with RACE?

s-step
Krylov
What else can we do with RACE?

- s-step Krylov
- Approx. Triangular solves
What else can we do with RACE?

- s-step Krylov
- Approx. Triangular solves
- Fancier Polynomial Preconditioners
What else can we do with RACE?

- s-step Krylov
- Approx. Triangular solves
- Fancier Polynomial Preconditioners
- Kaczmarz & CGMN
What else can we do with RACE?

- s-step Krylov
- Approx. Triangular solves
- Fancier Polynomial Preconditioners
- AMG Smoothers
- Kaczmarz & CGMN
What else can we do with RACE?

- s-step Krylov
- Approx. Triangular solves
- Fancier Polynomial Preconditioners
- AMG Smoothers
- Kaczmarz & CGMN
- Symmetric & transposed SpMV
What else can we do with RACE?

- s-step Krylov
- Approx. Triangular solves
- Fancier Polynomial Preconditioners
- AMG Smoothers

Alappat et al.: A Recursive Algebraic Coloring Technique for Hardware-efficient Symmetric Sparse Matrix-vector Multiplication, ACM TOPC 7 (3), 2020


Alappat et al.: Algebraic Temporal Blocking for Iterative Solvers on Multi-Core CPUs. To be submitted, will be on arXiv soon.
DCSE Summerschool next week: Linear Algebra on High Performance Computers

Few places available: https://www.aanmelder.nl/143287

Paolo Bientinesi (Umeå University)
Gerhard Wellein (U. Erlangen)
Laura Grigori (EPFL)