# Efficient Parallelization of MATLAB Stencil Applications for Multi-Core Clusters

**Johannes Spazier**, Steffen Christgau, Bettina Schnor

University of Potsdam, Germany

WOLFHPC 2016, Salt Lake City, USA

November 13, 2016

# Outline

# Outline

1. Introduction
2. Message Passing Interface
3. Hybrid Programming
4. Conclusion

# Introduction

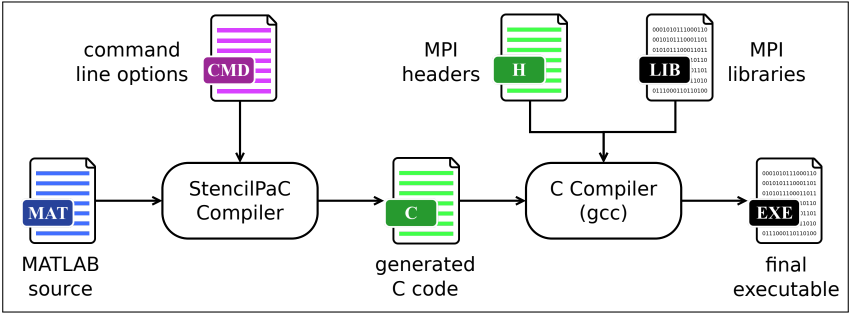
**MATLAB**

- approved as high-level language for scientific computing
- significantly reduced implementation effort
- enables fast prototyping of mathematical models
- well-suited for **stencil applications**
- drawback: slow execution through interpreter
- no out-of-the-box parallelization

$\Rightarrow$ insufficient performance for large data sets

## StencilPaC Overview

- MATLAB to parallel C compiler



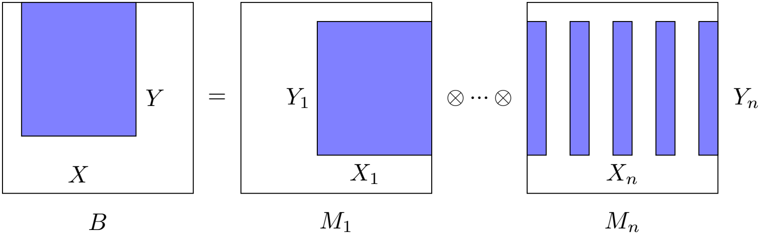
# Introduction

## StencilPaC Overview

- automatic parallelization for matrix operations

$$B(X, Y) = M_1(X_1, Y_1) \circ \ldots \circ M_n(X_n, Y_n)$$

- support different architectures
  - shared and distributed memory systems, accelerators
- build on common programming APIs
  - OpenMP, MPI and OpenACC



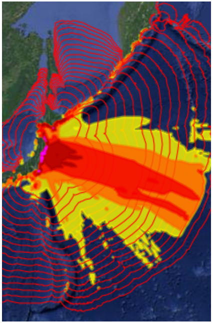
# Introduction

### Applications

- two grid-based stencil applications
- domain update over multiple iterations
- manual reference implementations in C/C++

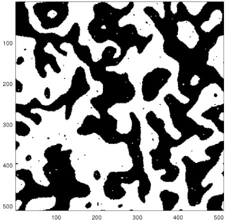
### EasyWave

- tsunami simulation developed at the German Research Center for Geosciences
- access pattern: 5-point-stencil



### Cellular Automaton

- idealized model for biological systems
- 9-point-stencil (moore neighborhood)



# Introduction

**StencilPaC Overview**

- generated C code is much faster than MATLAB for both applications
- improvements of more than
  - **7 times** with sequential code
  - **21 times** on an 8 core shared memory system
  - **187 times** with an NVIDIA Tesla K40m

  for the memory-bound tsunami simulation *EasyWave*
- even better results for the *Cellular Automaton*

# Introduction

## StencilPaC Overview

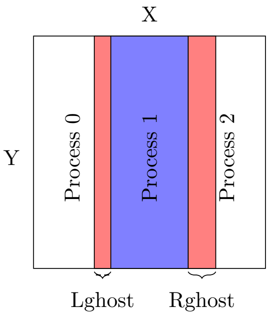
- distributed systems are most challenging
  - automatic partitioning of matrices
  - generic handling of communication between processes
  - partial computation
- small runtime overhead is essential
- focus on MPI one-sided API in previous work
- **today:** concepts of and comparison with
  - two-sided communication
  - hybrid programming

# Outline

# Message Passing Interface

## Principles

- degree of parallelization is given by the number of processes
- distribute matrices evenly among the processes
- one-dimensional domain decomposition (block of columns)
- compute local parts in parallel
- set up communication at runtime
- provide appropriate ghost zones



## Distributed Computation

- choose base matrix $B$

  $$B(X, Y) = M_1\,(X_1, Y_1) \circ \ldots \circ M_n\,(X_n, Y_n)$$

- compute local part of $B$

```
for (j = 0; j < length(X); j++) {
  if (is_local( B, X(j) )) {
    for (k = 0; k < length(Y); k++) {
      B( X(j), Y(k) ) = M1 ( X1 (j), Y1 (k) )
                        o ...
                        o Mn ( Xn (j), Yn (k) );
    }
  }
}
```

# Message Passing Interface

## 1. One-sided Communication

- direct access on remote memory with `MPI_Get`
- ghost zones can be fetched without involving other processes
- ranks calculated based on equally sized partitioning
- coarse-grained synchronization with `MPI_Win_fence`

---

⇒ simple API for generic data exchange

⇒ less administration at runtime

⇒ expensive synchronization

# One-sided Communication

## Generic Data Exchange

$$B(X, Y) = M_1(X_1, Y_1) \circ \ldots \circ M_n(X_n, Y_n)$$

```
MPI_Win_fence( 0, Mi.win );

for (j = 0; j < length(X); j++) {
  if (is_local( B, X(j))) {
    if (!is_local(M1, X1(j)))
      MPI_Get(M1.win, X1(j) ...);
    ...
    if (!is_local(Mn, Xn(j)))
      MPI_Get(Mn.win, Xn(j) ...);
  }
}

MPI_Win_fence( 0, Mi.win );
```

# Message Passing Interface

## 2. Two-sided Communication

- exchange ghost zones via messages
- both sender and receiver are involved
- send operations must also be provided
- use non-blocking operations to avoid deadlocks (`MPI_ISend` and `MPI_IRecv`)
- synchronize with `MPI_Waitall`

---

⇒ pair-wise synchronization

⇒ additional administration required

# Two-sided Communication

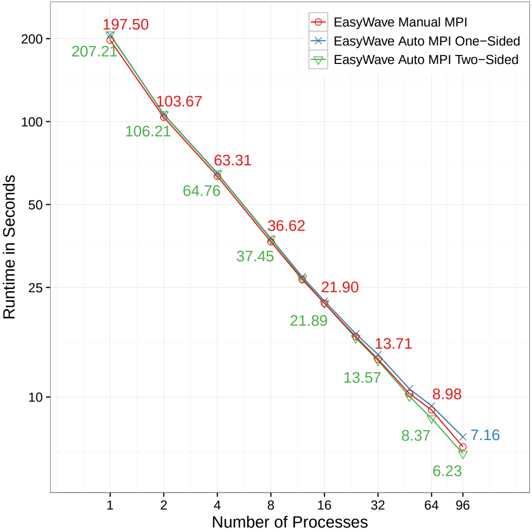
## Generic Data Exchange

$$B(X,Y) = M_1(X_1,Y_1) \circ \ldots \circ M_n(X_n,Y_n)$$

```
for (j = 0; j < length(X); j++) {
  if (is_local( B, X(j)))
    if (!is_local(M1, X1(j)))
      MPI_Irecv(M1.vdata, X1(j), ...);

  if (is_local(M1, X1(j)))
    if (!is_local(B, X(j)))
      MPI_Isend(M1.vdata, X1(j), ...);

  /* Repeat for other matrices. */
}

MPI_Waitall( Mi.reqnr, Mi.requests, ...);
Mi.reqnr = 0;
```

# Evaluation: One- vs. two-sided

## EasyWave




## Platform

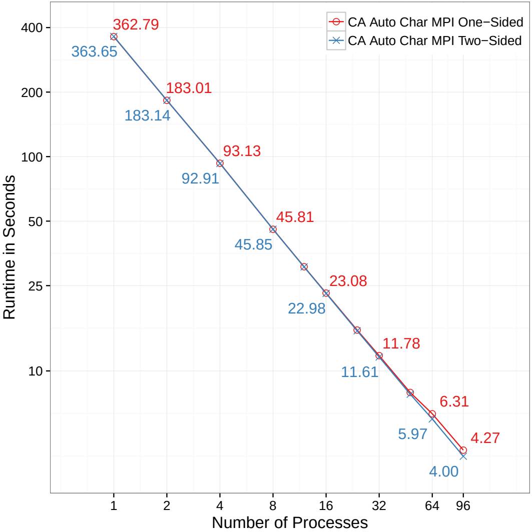
- 12 dual-socket nodes
- 4-core Intel Xeon CPUs
- InfiniBand Network
- Open MPI 1.8.2 and GCC 4.9.1

## Results

- generated codes can keep up with hand-written one
- similar scaling of all versions
- two-sided is 13% faster than one-sided

# Evaluation: One- vs. two-sided

## Cellular Automaton




## Results

- overall adequate scaling
- speedup of at least 85 on 96 cores
- 6% improvement with two-sided version

# Evaluation: One- vs. two-sided

**Summary**

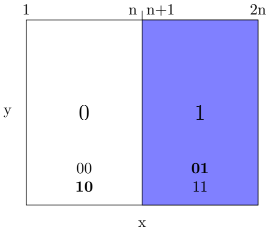
- similar findings for both applications
- satisfying scaling even for larger core counts
- runtime of hand-written codes almost reached
- two-sided MPI implementation performs better than one-sided
  - despite higher runtime overhead
  - benefiting from fine-grained synchronization

# Outline

# Hybrid Programming

## Two-sided MPI + OpenMP

- each MPI process spawns multiple threads
- work is divided statically among these threads
- simple combination leads to serious load imbalances
- process distribution and thread partitioning interfere
- amount of computational work varies




```
#pragma omp parallel for private(k)
for (j = 0; j < length(X); j++)
  if (is_local( B, X(j) ))
    for (k = 0; k < length(Y); k++)
      B( X(j), Y(k) ) = ...
```

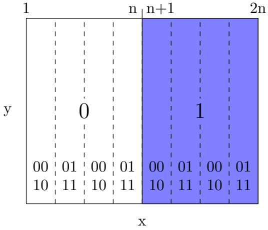
# Hybrid Programming

## 1. Dynamic Scheduling

- use dynamic scheduling of threads
- chunks are assigned at runtime
- better sharing of computational work expected
- easy implementation with

```
#pragma omp parallel schedule(dynamic)
```

- additional runtime overhead

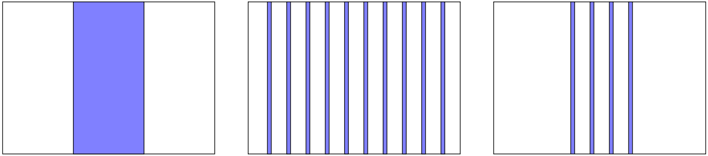


## 2. Intersection Approach

- optimization for special matrix access
- based on MATLAB's range index

```
start:step:end = [start, start+step, ..., end]
```

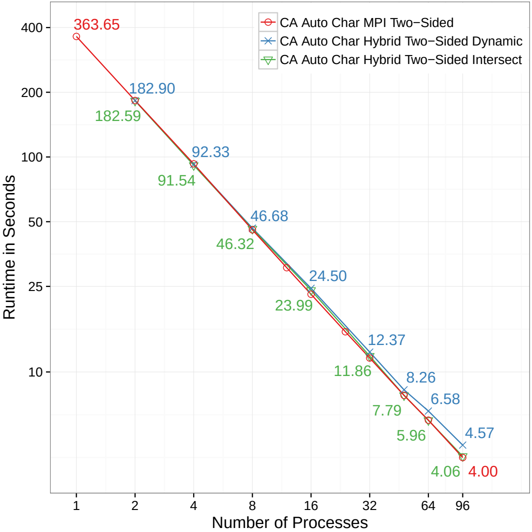
- local portion can be determined in advance
- no locality check at runtime anymore
- enables static thread scheduling again



Local matrix portion | Global index range | Local index range

# Evaluation: Hybrid Programming

## Cellular Automaton

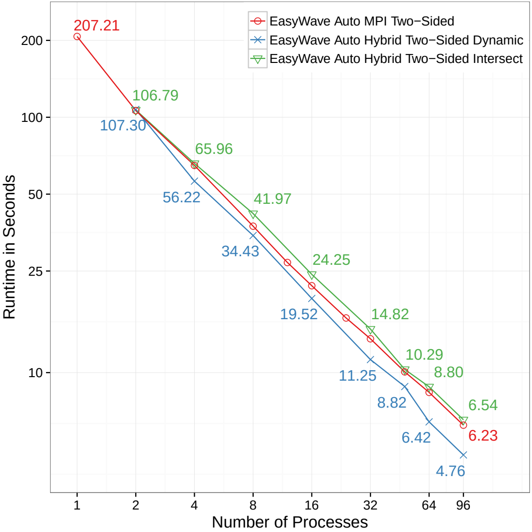



## Results

- pure MPI version performs best
- similar results with hybrid intersection
- overhead of 14% with dynamic scheduling

# Evaluation: Hybrid Programming

## EasyWave




### Results

- pure MPI better than hybrid intersection
- dynamic approach outperforms other versions
- improvement of 24%
- better load balancing for memory-bound applications

# Outline

# Conclusion

Successful extension of StencilPaC for multi-core clusters.

## Message Passing Interface

- well suited for automatic parallelization
- slightly better results with two-sided communication
- speedups of up to 91 on 96 cores

## Hybrid Programming

- benefit of hybrid versions depends on application demands
- dynamic scheduling can reduce load imbalances
  - improvement of 24% on a memory-bound simulation
- intersection approach does not show any benefit

# Conclusion

## Future Work

- examine a wider range of applications
- consider additional platforms
  - use other MPI implementations (e.g. Open MPI 2.0)
  - compare different architectures and network types
- deeper analysis of observed effects in hybrid programming
- evaluate possible use of generated C code on FPGAs

# Questions?

**Thanks for**
**your attention.**