# Research Presentation for Computer Modelling Group

**Malcolm Roberts**

University of Strasbourg

2016-04-26

malcolm.i.w.roberts@gmail.com, www.malcolmiwroberts.com

# Outline

- Convolutions
  - Implicitly Dealiased FFT-based convolutions
  - Shared-memory implementation
  - Parallel `OpenMP`/`MPI` implementation
  - Pseudospectral simulations
- `GPU` programming
  - `OpenCL`
  - `schnaps`
  - Performance analysis.

# FFT-based convolutions

The convolution of $\{F_k\}_{k=0}^{m-1}$ and $\{G_k\}_{k=0}^{m-1}$ is

$$(F \star G)_k = \sum_{\ell=0}^{k} F_\ell G_{k-\ell}, \quad k=0, \ldots, m-1. \tag{1}$$

Applications:

- Signal processing
- Machine learning: convolutional neural networks
- Image processing
- Particle-Image-Velocimitry
- Pseudospectral simulations of nonlinear PDEs

Using FFTs improves speed and accuracy.

# FFT-based convolutions

The convolution theorem:

$$\mathcal{F}\left[F * G\right] = \mathcal{F}\left[F\right] \odot \mathcal{F}\left[G\right]. \tag{2}$$

Let $\zeta_m = \exp\left(\frac{2\pi i}{m}\right)$. Forward and backward Fourier transforms are given by:

$$f_j = \sum_{k=0}^{m-1} \zeta_m^{jk} F_k, \qquad F_k = \frac{1}{m} \sum_{j=0}^{m-1} \zeta_m^{-kj} f_k, \tag{3}$$

We will use the identity

$$\sum_{j=0}^{m-1} \zeta_m^{\ell j} = \begin{cases} m & \text{if } \ell = sm \text{ for } s \in \mathbb{Z}, \\ \frac{1-\zeta_m^{\ell m}}{1-\zeta_m^m} = 0 & \text{otherwise.} \end{cases} \tag{4}$$

# FFT-based convolutions

The convolution theorem works because

$$\begin{aligned}\sum_{j=0}^{m-1} f_j g_j \zeta_m^{-jk} &= \sum_{j=0}^{m-1} \zeta_m^{-jk} \left(\sum_{p=0}^{m-1} \zeta_m^{jp} F_p\right) \left(\sum_{q=0}^{m-1} \zeta_m^{jq} G_q\right) \\ &= \sum_{p=0}^{m-1} F_p \sum_{q=0}^{m-1} G_q \sum_{j=0}^{m-1} \zeta_m^{j(-k+p+q)} \qquad (5) \\ &= m \sum_{s} \sum_{p=0}^{m-1} F_p G_{k-p+sm}.\end{aligned}$$

The terms $s \neq 0$ are aliases; they are bad.

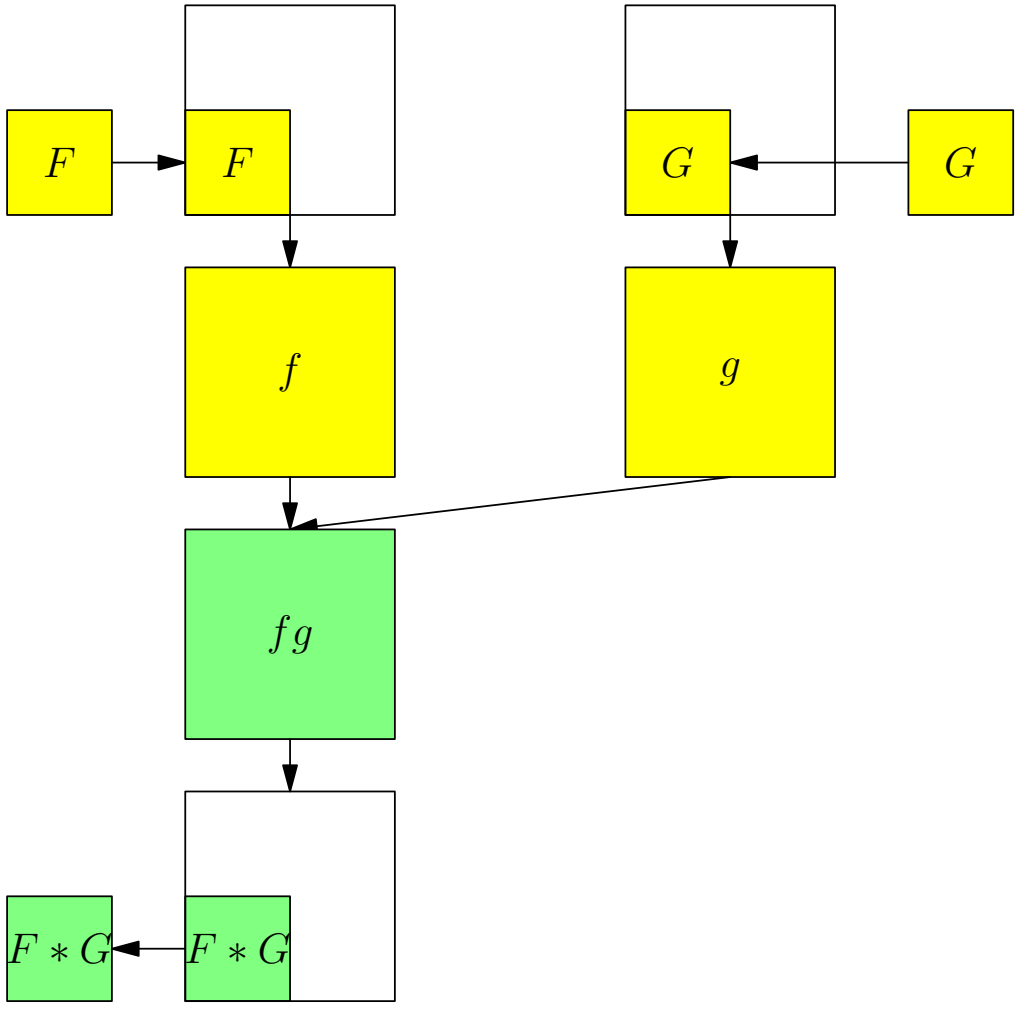
# Conventional dealiasing: zero padding

Let $\widetilde{F} \doteq \{F_0, F_1, \ldots, F_{m-2}, F_{m-1}, \underbrace{0, \ldots, 0}_{m}\}$. Then,

$$\begin{aligned}
\left(\widetilde{F} *_{2m} \widetilde{G}\right)_k &= \sum_{\ell=0}^{2m-1} \widetilde{F}_{\ell \bmod (2m)} \widetilde{G}_{(k-\ell) \bmod (2m)} \\
&= \sum_{\ell=0}^{m-1} F_\ell \widetilde{G}_{(k-\ell) \bmod (2m)} \\
&= \sum_{\ell=0}^{k} F_\ell G_{k-\ell}.
\end{aligned} \tag{6}$$

There is also a "2/3" padded version for pseudospectral simulations, where the input $\{F_k\}_{k=-m}^{m-1}$ is padded to $3m$.

# Dealiasing with conventional zero-padding



# Dealiasing with implicit zero-padding

We modify the FFT to account for the zeros implicitly.

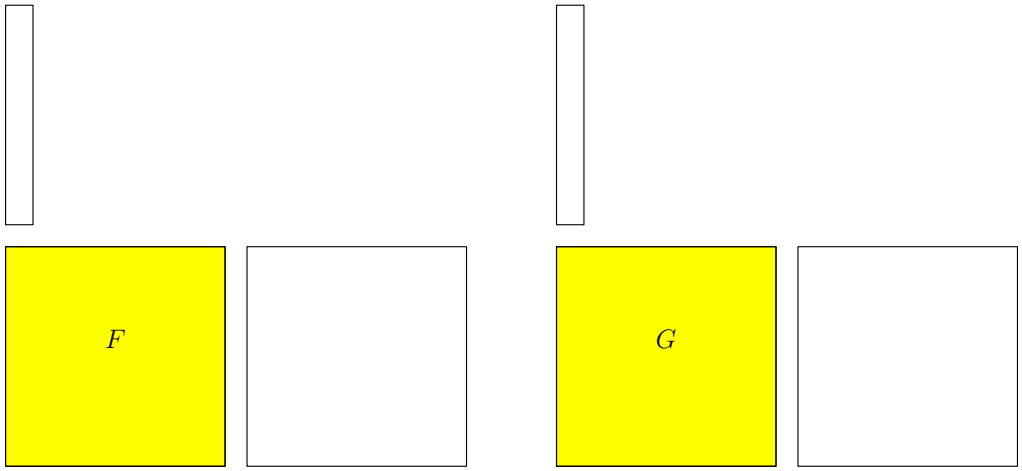
Let $\zeta_n = \exp(-i2\pi/n)$. The Fourier transform of $\widetilde{F}$ is

$$f_x = \sum_{k=0}^{2m-1} \zeta_{2m}^{xk} \widetilde{F}_k = \sum_{k=0}^{m-1} \zeta_{2m}^{xk} \widetilde{F}_k \tag{7}$$

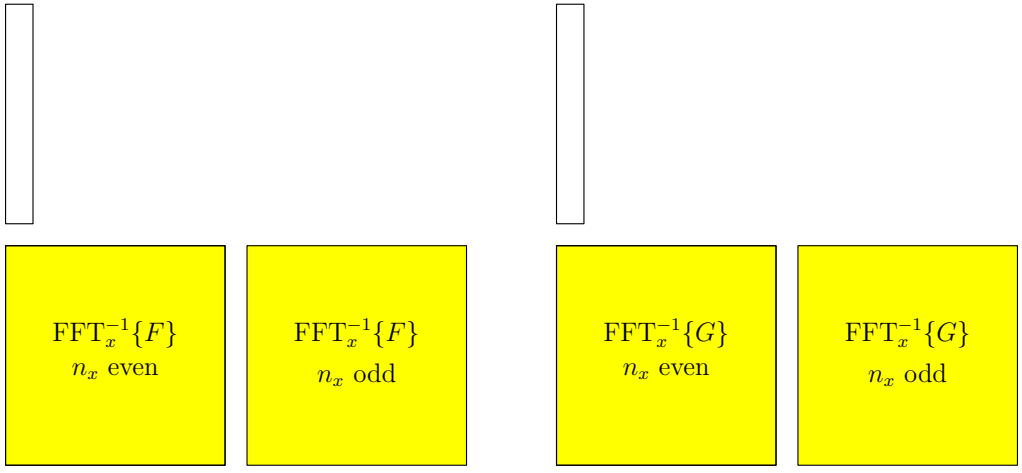
We can compute this using two discontiguous buffers:

$$f_{2x} = \sum_{k=0}^{m-1} \zeta_m^{xk} F_k \quad f_{2x+1} = \sum_{k=0}^{m-1} \zeta_m^{xk} \left(\zeta_{2m}^{k} F_k\right). \tag{8}$$

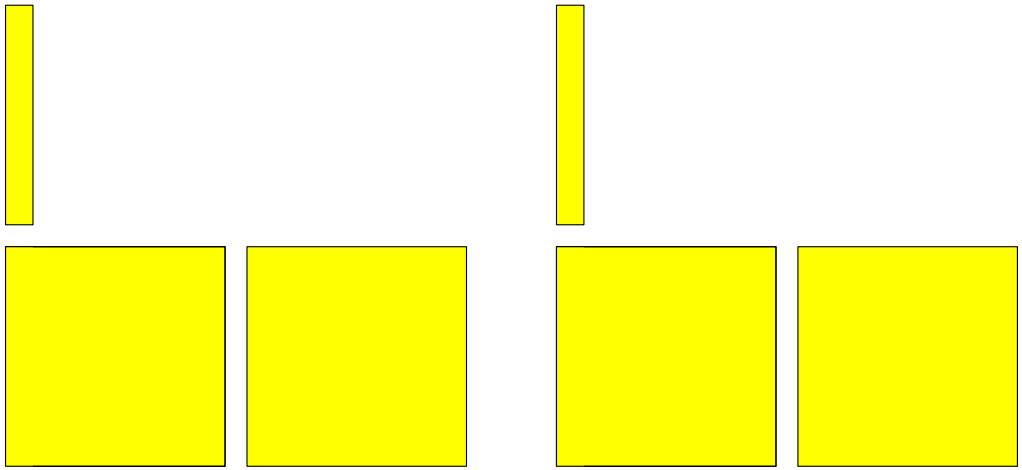
# Dealiasing with implicit zero-padding



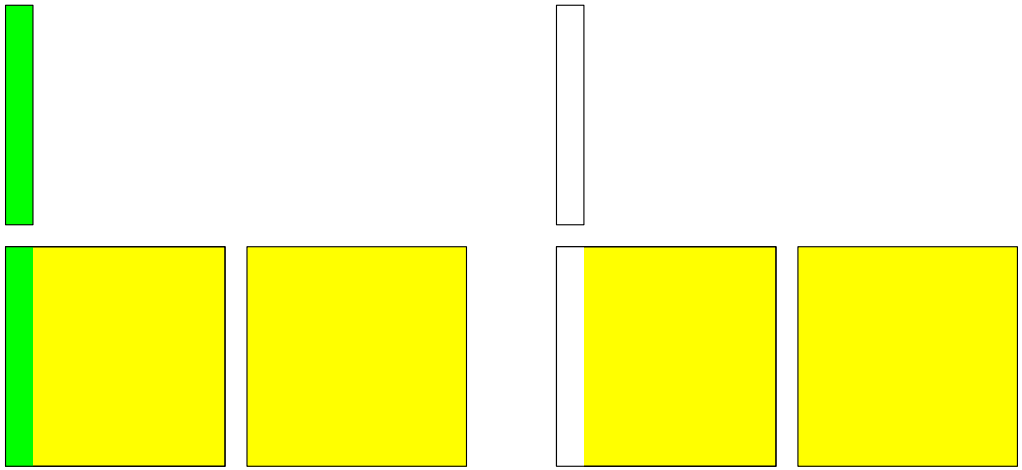
# Dealiasing with implicit zero-padding



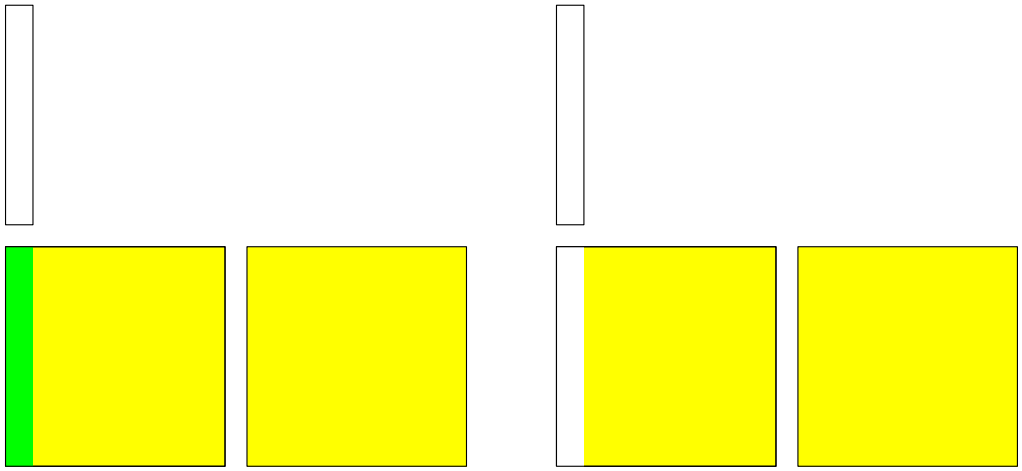
# Dealiasing with implicit zero-padding



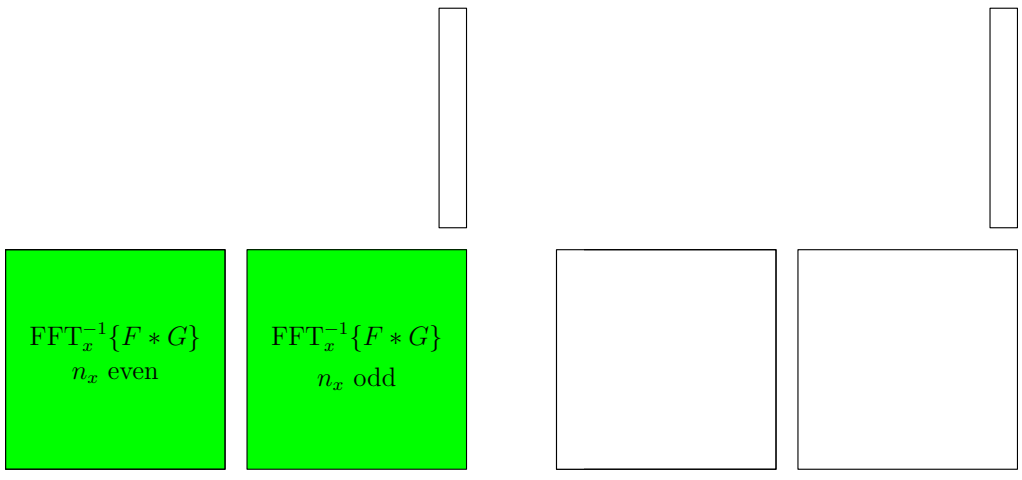
# Dealiasing with implicit zero-padding



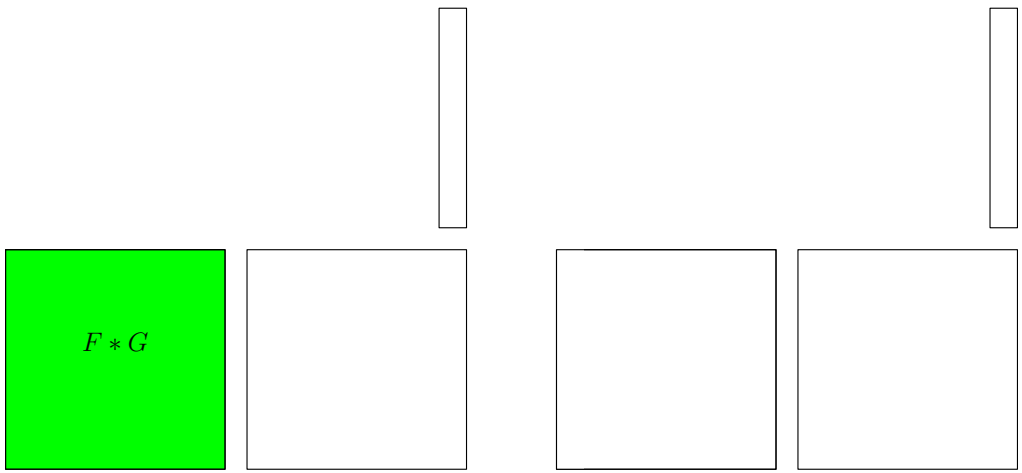
# Dealiasing with implicit zero-padding



# Dealiasing with implicit zero-padding



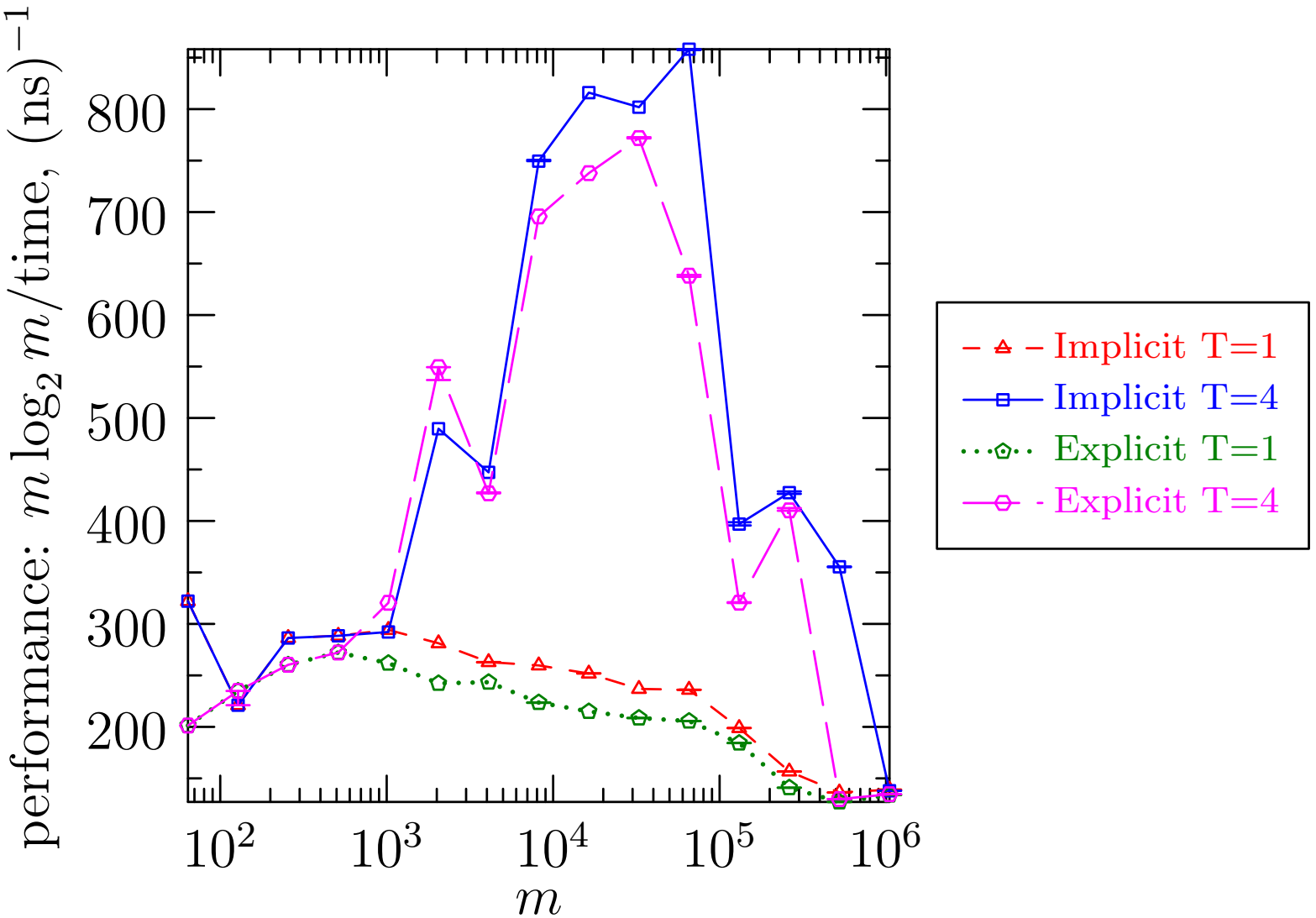
# Dealiasing with implicit zero-padding



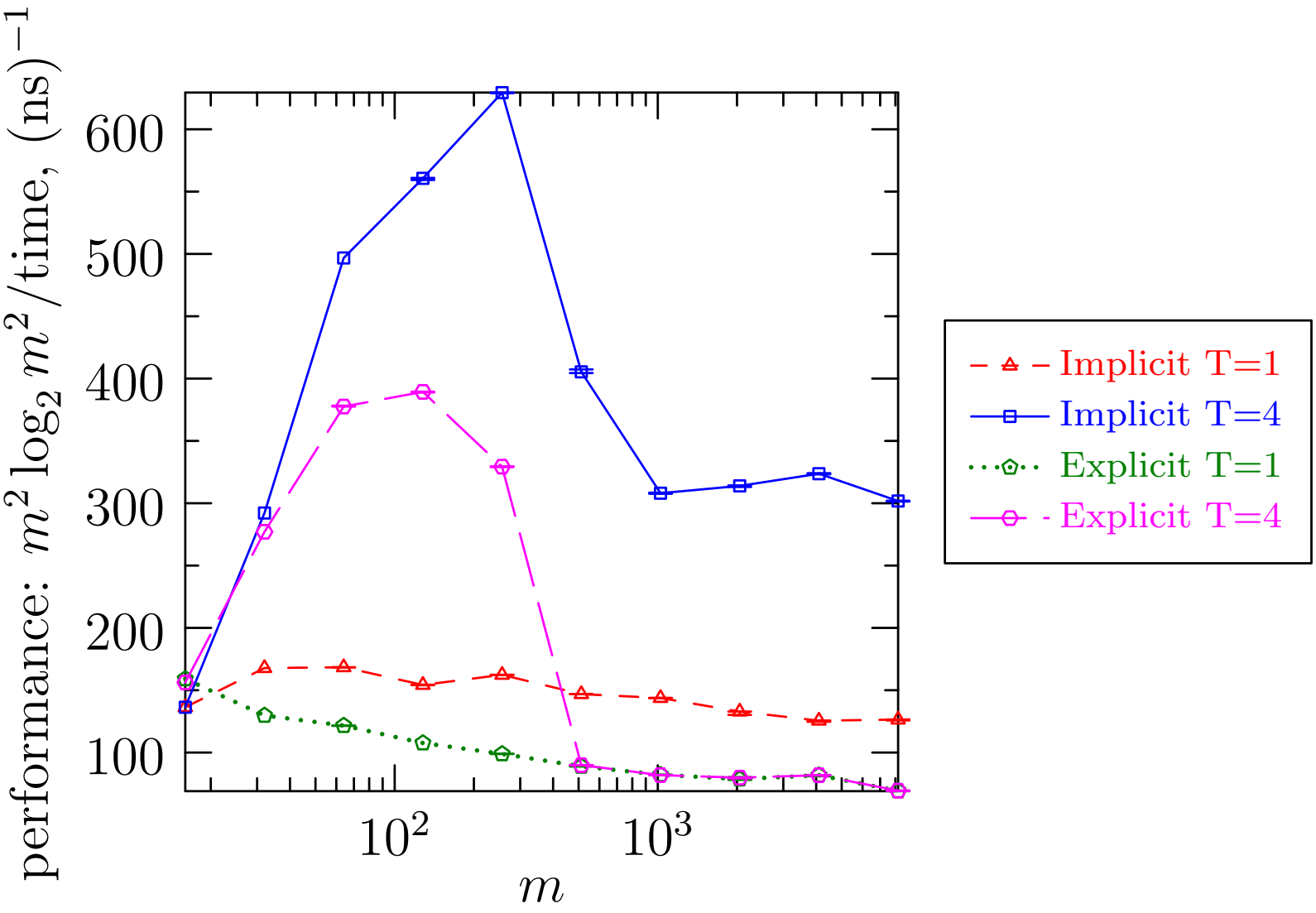
# Shared-memory implementation

- Implicit dealiasing requires less memory.
- We avoid FFTs on zero-data.
- By using discontiguous buffers, we can use multiple NUMA nodes.
- SSE2 vectorization instructions.
- Additional threads requires additional sub-dimensional work buffers.
- We use strides instead of transposes because we need to multi-thread.

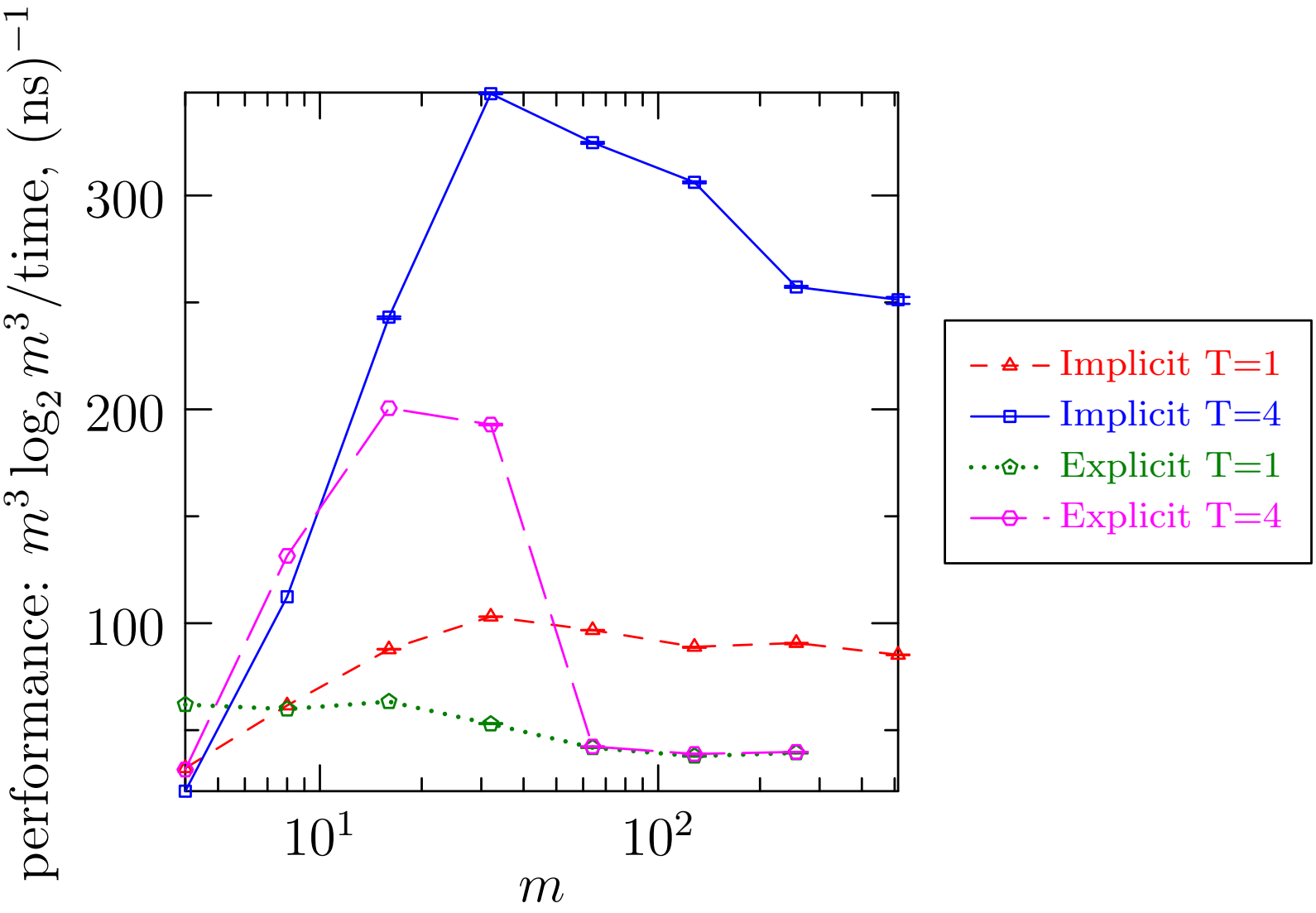
# Multi-threaded performance: 1D



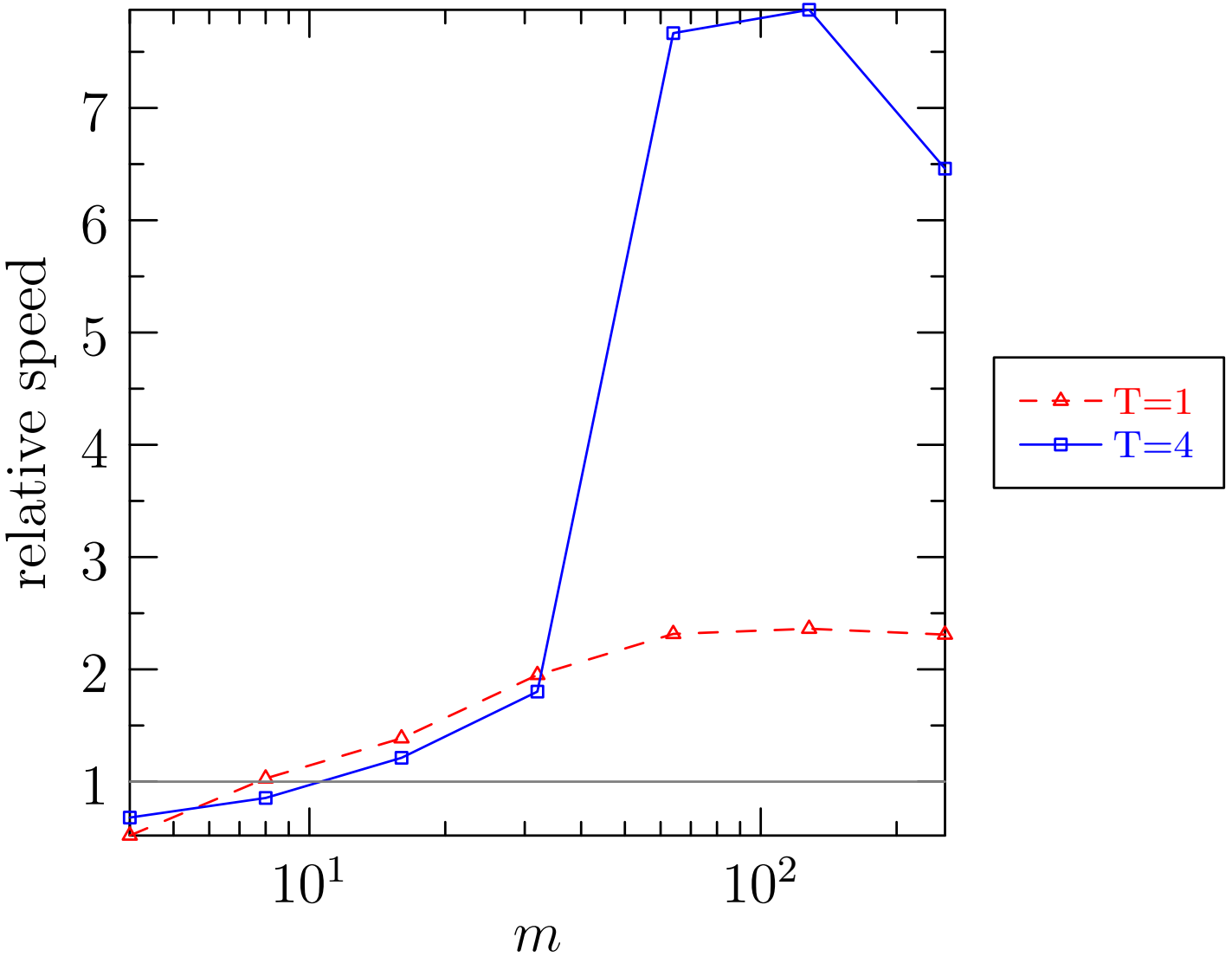
# Multi-threaded performance: 2D



# Multi-threaded performance: 3D



# Multi-threaded speedup: 3D



# Distributed-memory implementation

- Implicit dealiasing requires less communication.
- By using discontiguous buffers, we can overlap communication and computation.
- We use a hybrid `OpenMP`/`MPI` parallelization for clusters of multi-core machines.
- 2D `MPI` data decomposition.
- We make use of the *hybrid transpose* algorithm.

# Hybrid MPI Transpose

Matrix transpose is an essential primitive of high-performance computing.

They allow one to localize data on one process so that shared-memory algorithms can be applied.

I will discuss two algorithms for transposes:

- Direct Transpose.
- Recursive Transpose.

We combine thses into a *hybrid transpose*.

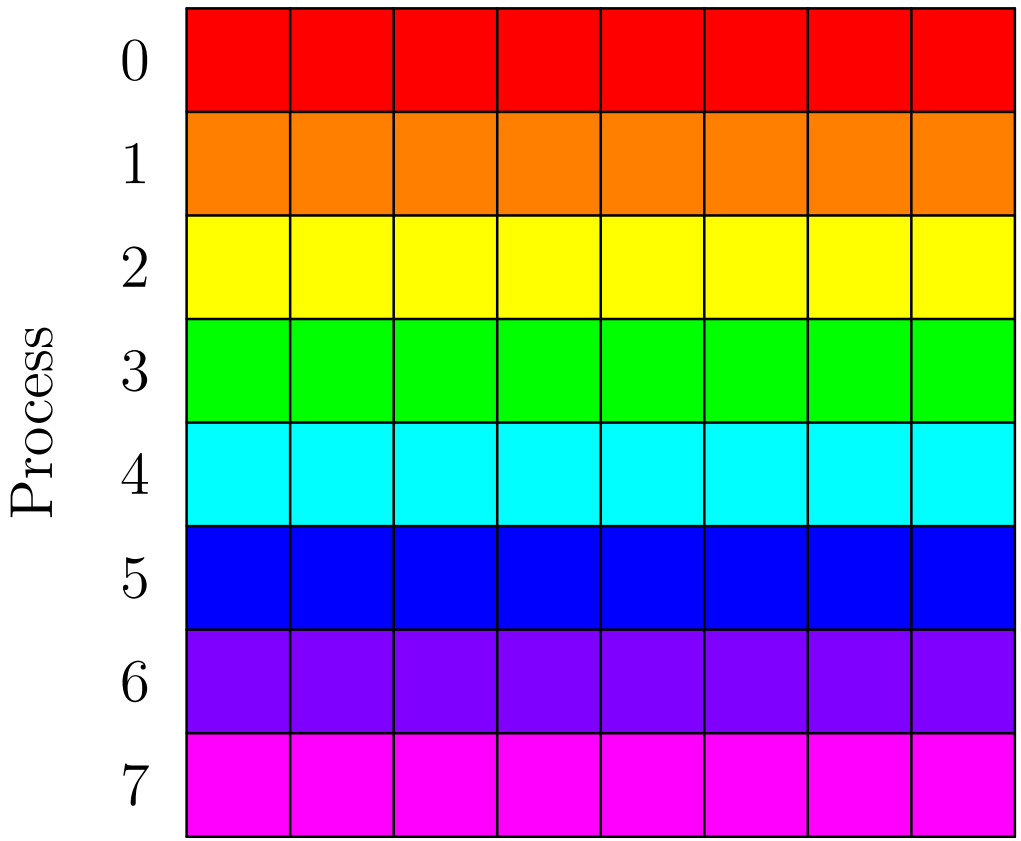
# Direct (AlltoAll) Transpose

- Efficient for $P \gg m$ (large messages).
- Most direct method.
- Many small messages when $P \approx m$.

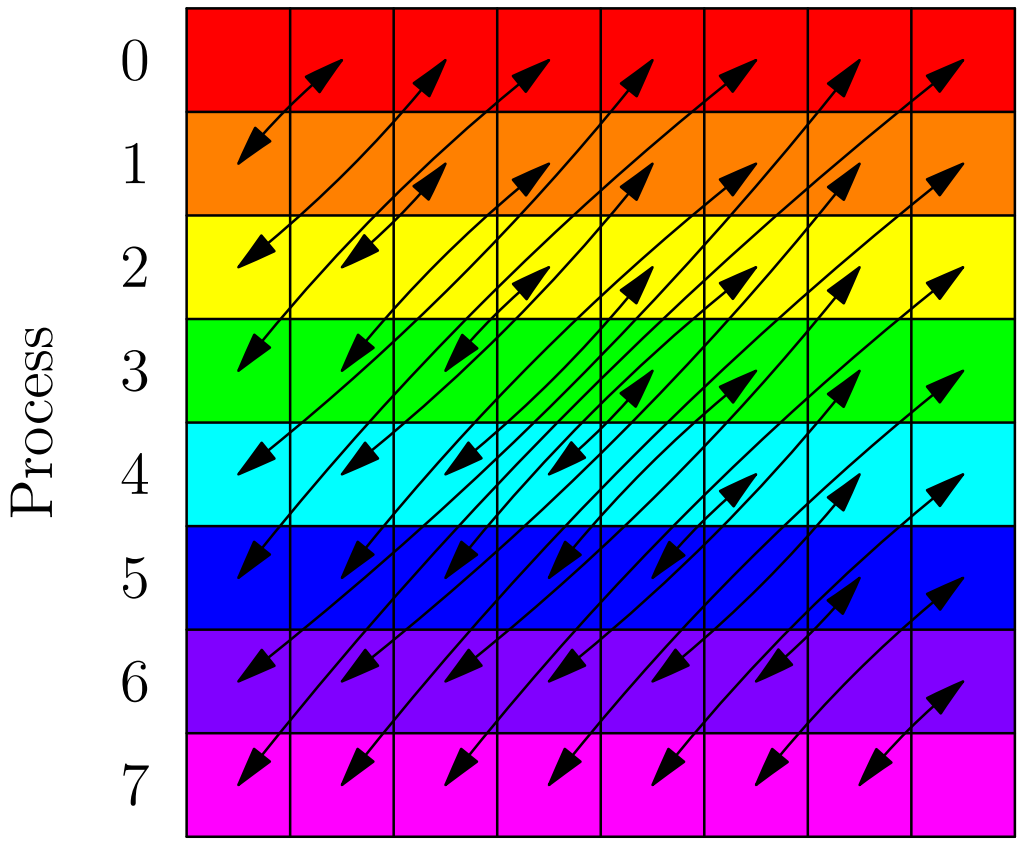
Implementations:

- `MPI_Alltoall`
- `MPI_Send`, `MPI_Recv`

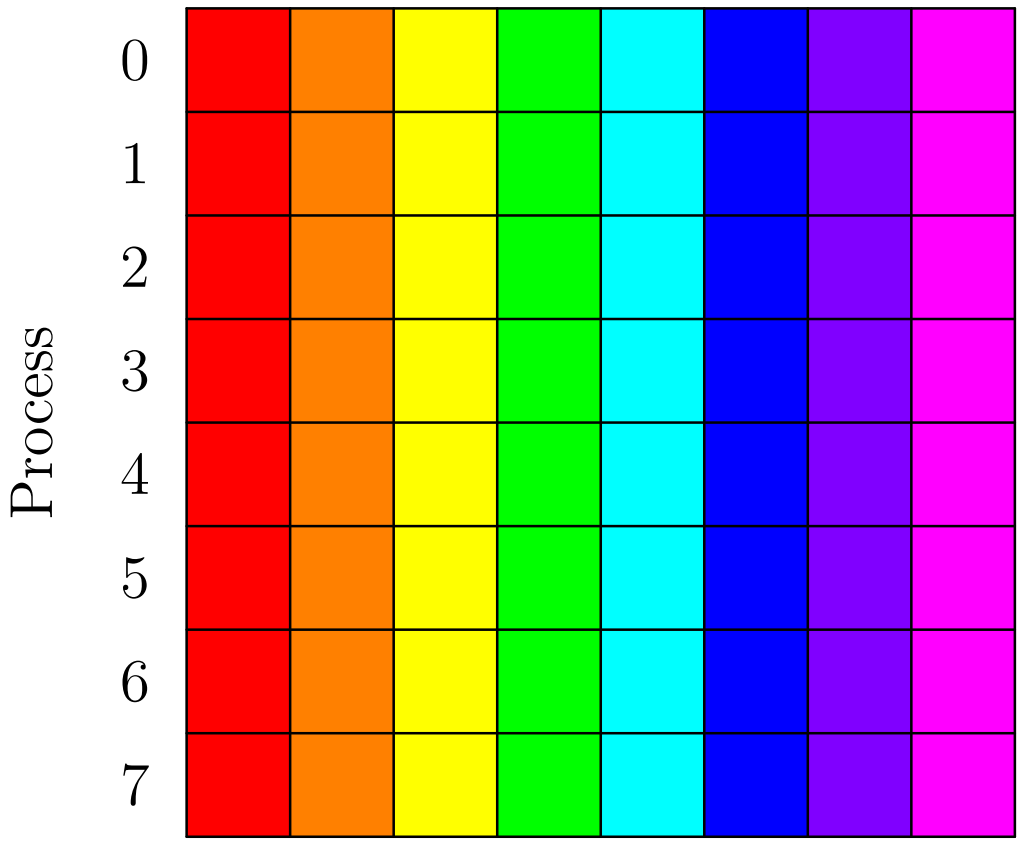
# Direct (AlltoAll) Transpose



# Direct (AlltoAll) Transpose



# Direct (AlltoAll) Transpose



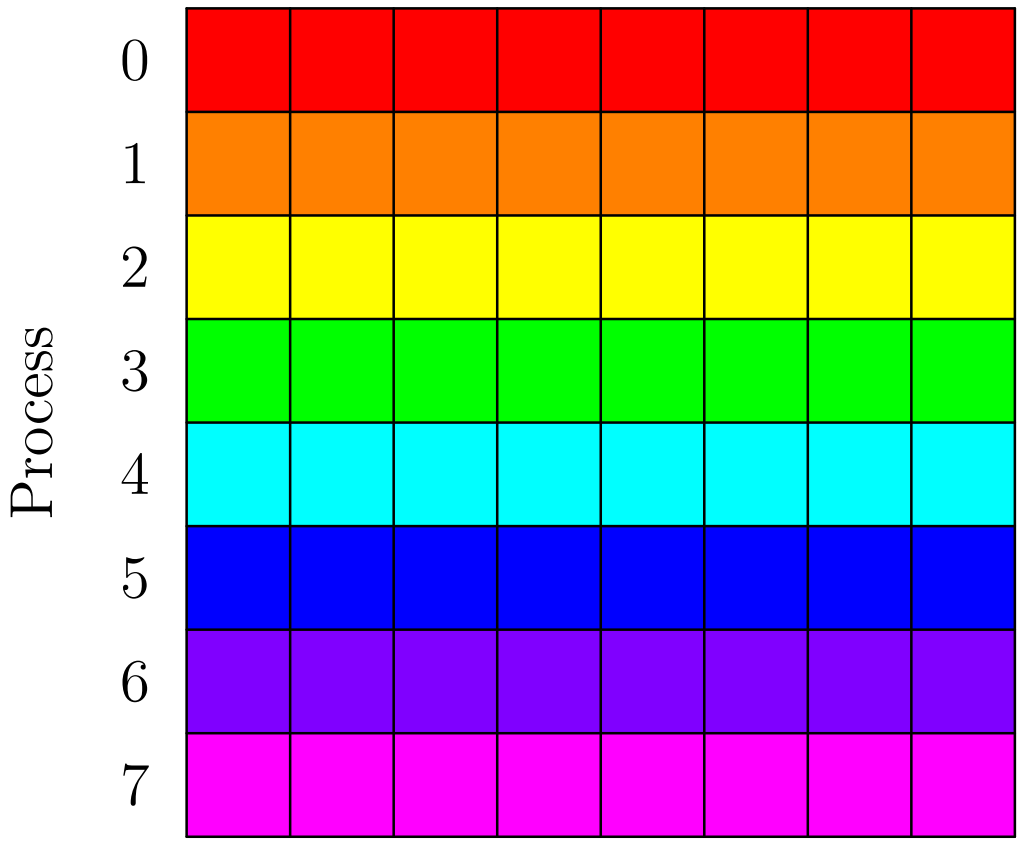
# Recursive Transpose

- Efficient for $P \ll m$ (large messages).
- Recursively subdivides transpose into smaller block transposes.
- $\log m$ phases.
- Communications are grouped to reduce latency.
- Requires intermediate communication.

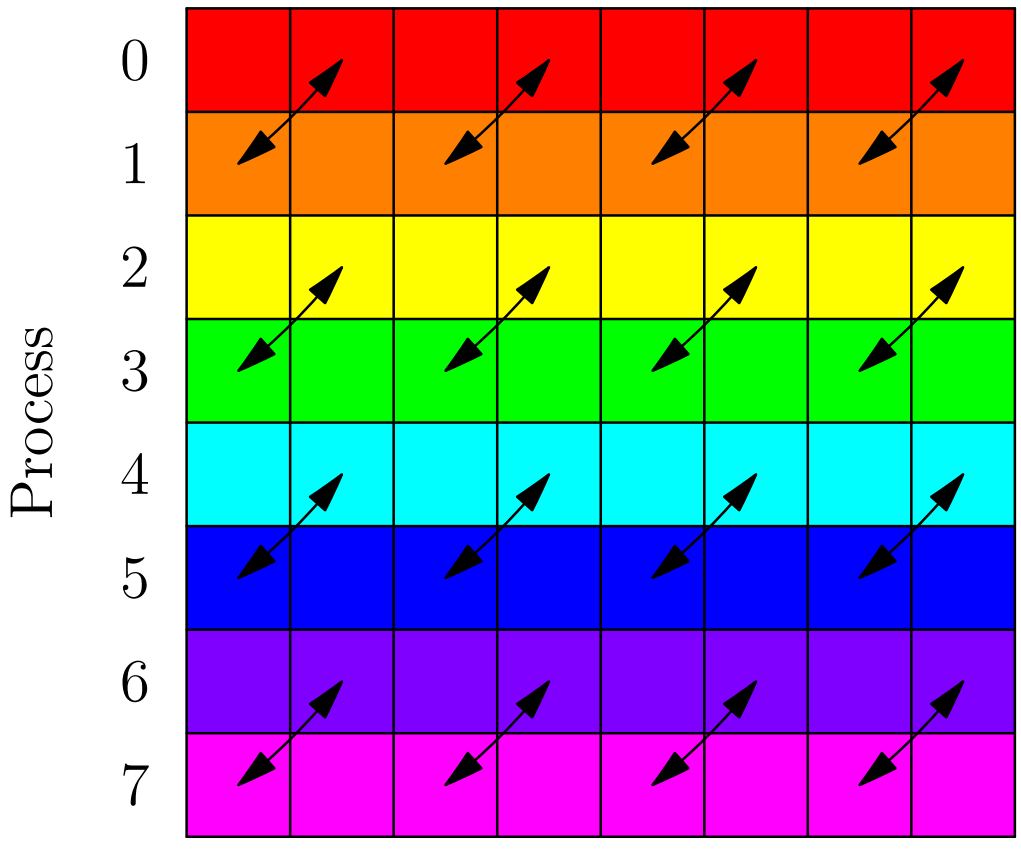
Implementations:

- `FFTW`

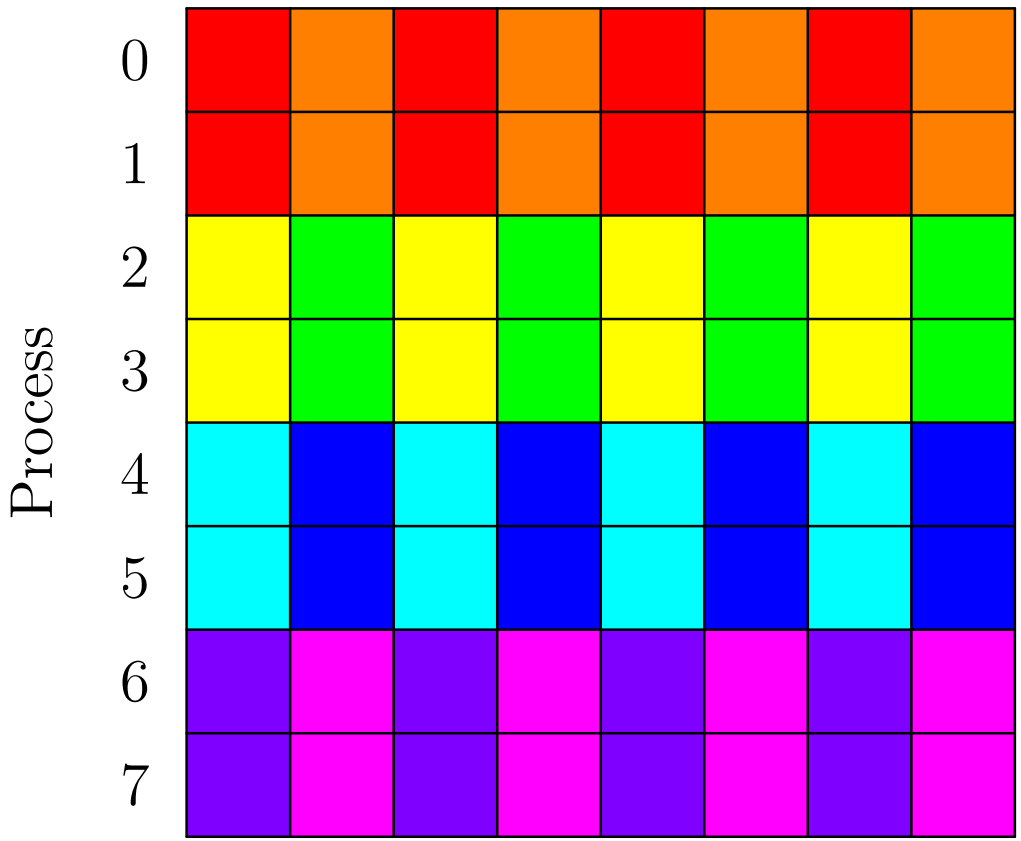
# Recursive Transpose



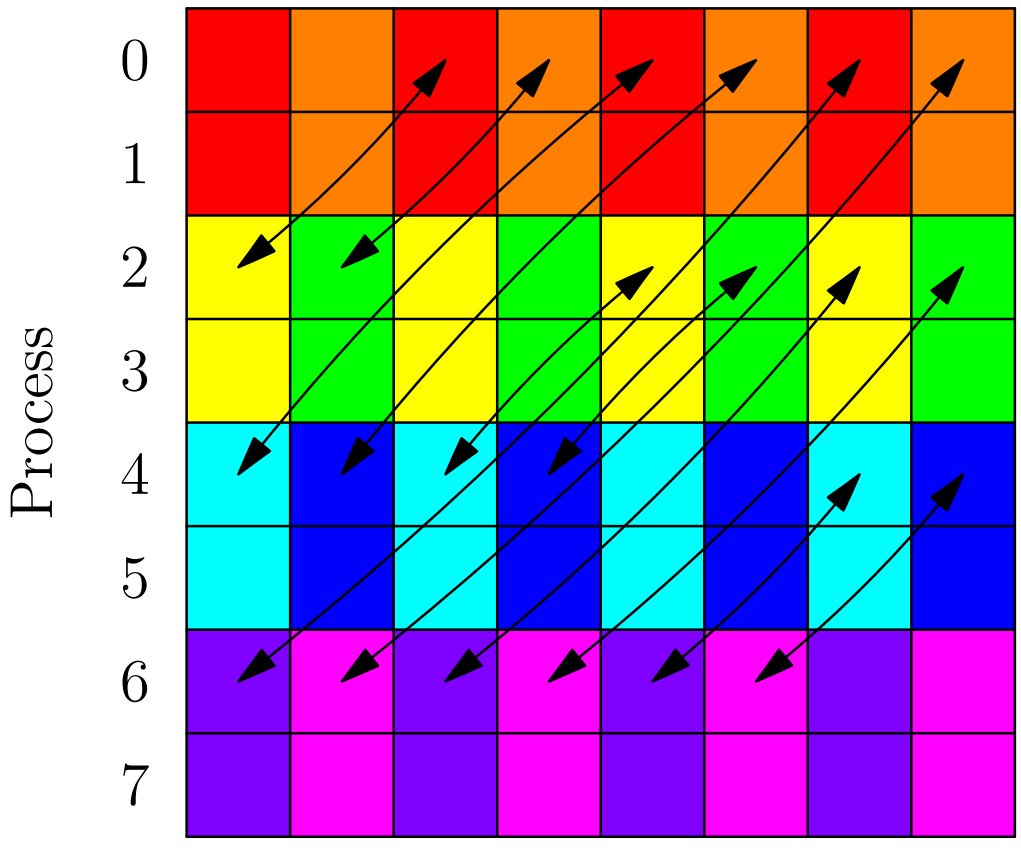
# Recursive Transpose



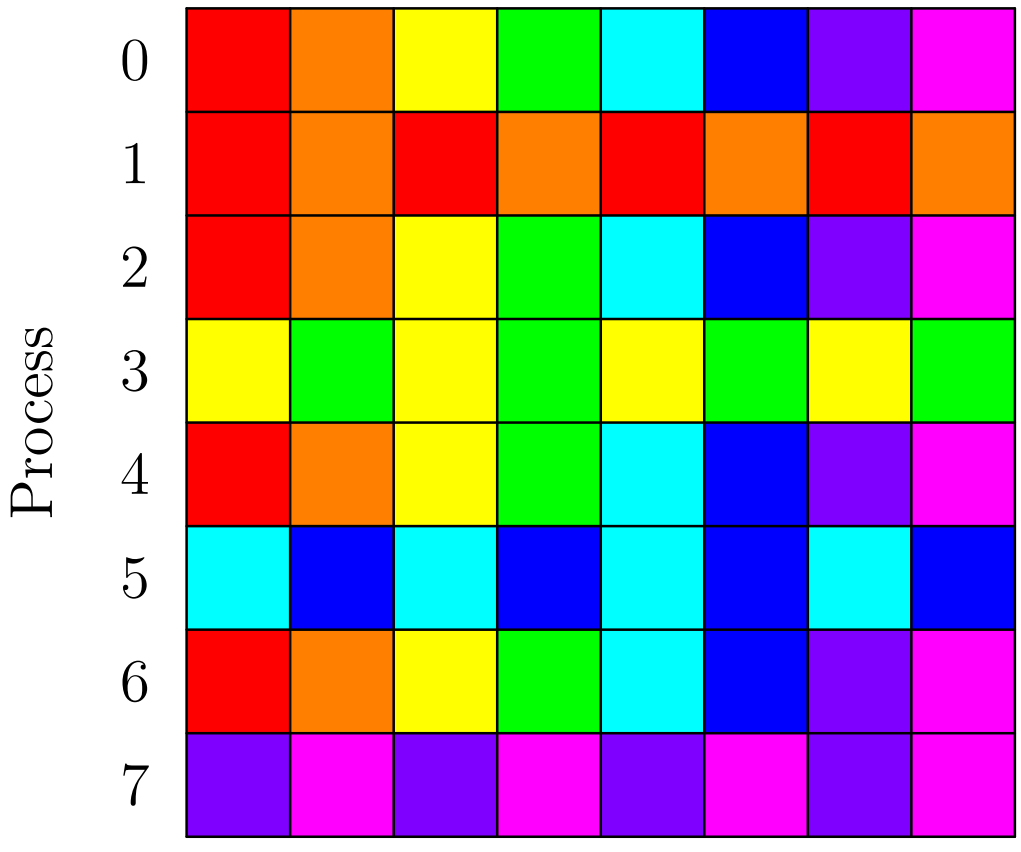
# Recursive Transpose



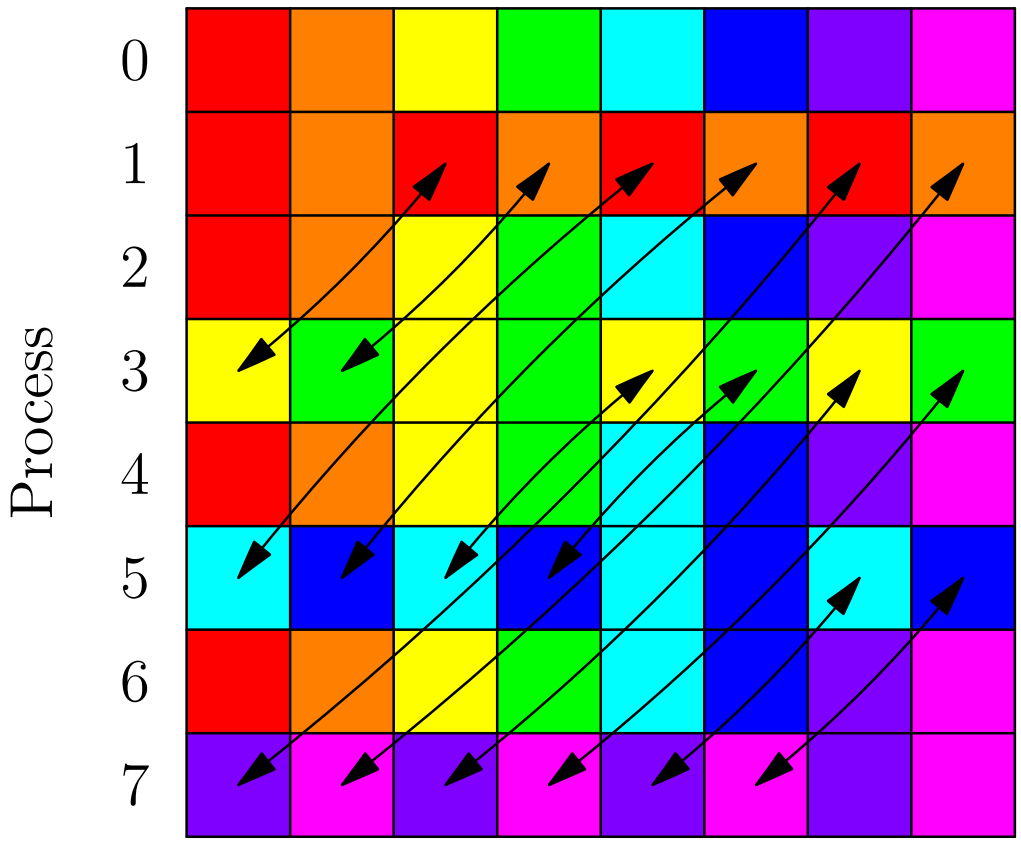
# Recursive Transpose



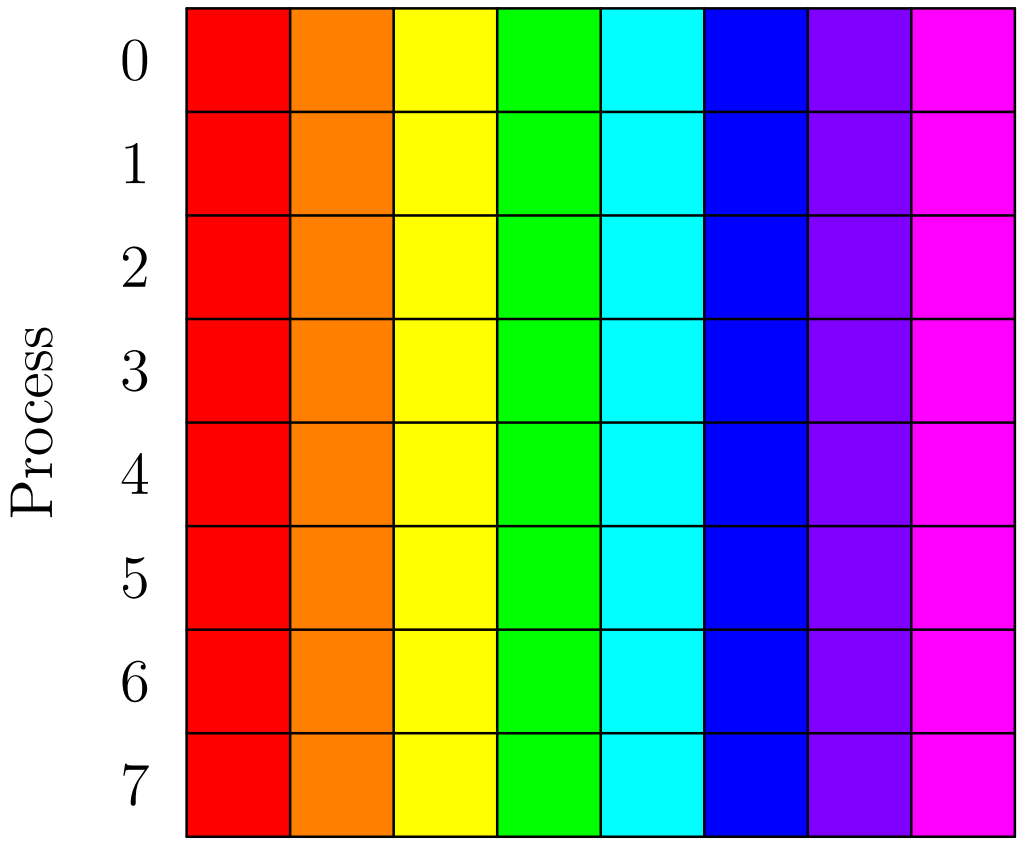
# Recursive Transpose



# Recursive Transpose



# Recursive Transpose



# Hybrid Transpose

- Recursive, but just one level.
- Use the empirical properties of the cluster to determine best parameters.
- *Optionally* group messages to reduce latency.

Implementation:

- `FFTW++`

Direct transpose communication cost: $\frac{P-1}{P^2}m^2$, $P$ messages.

Hybrid cost with $P = ab$: $\frac{(a-1)bm^2}{P^2} + \frac{(b-1)am^2}{P^2}$, $a + b$ messages.

# Hybrid Transpose

Let $\tau_\ell$ be the message latency, and $\tau_d$ the time to send one element. The time to send $n$ elements is

$$\tau_\ell + n\tau_d. \tag{9}$$

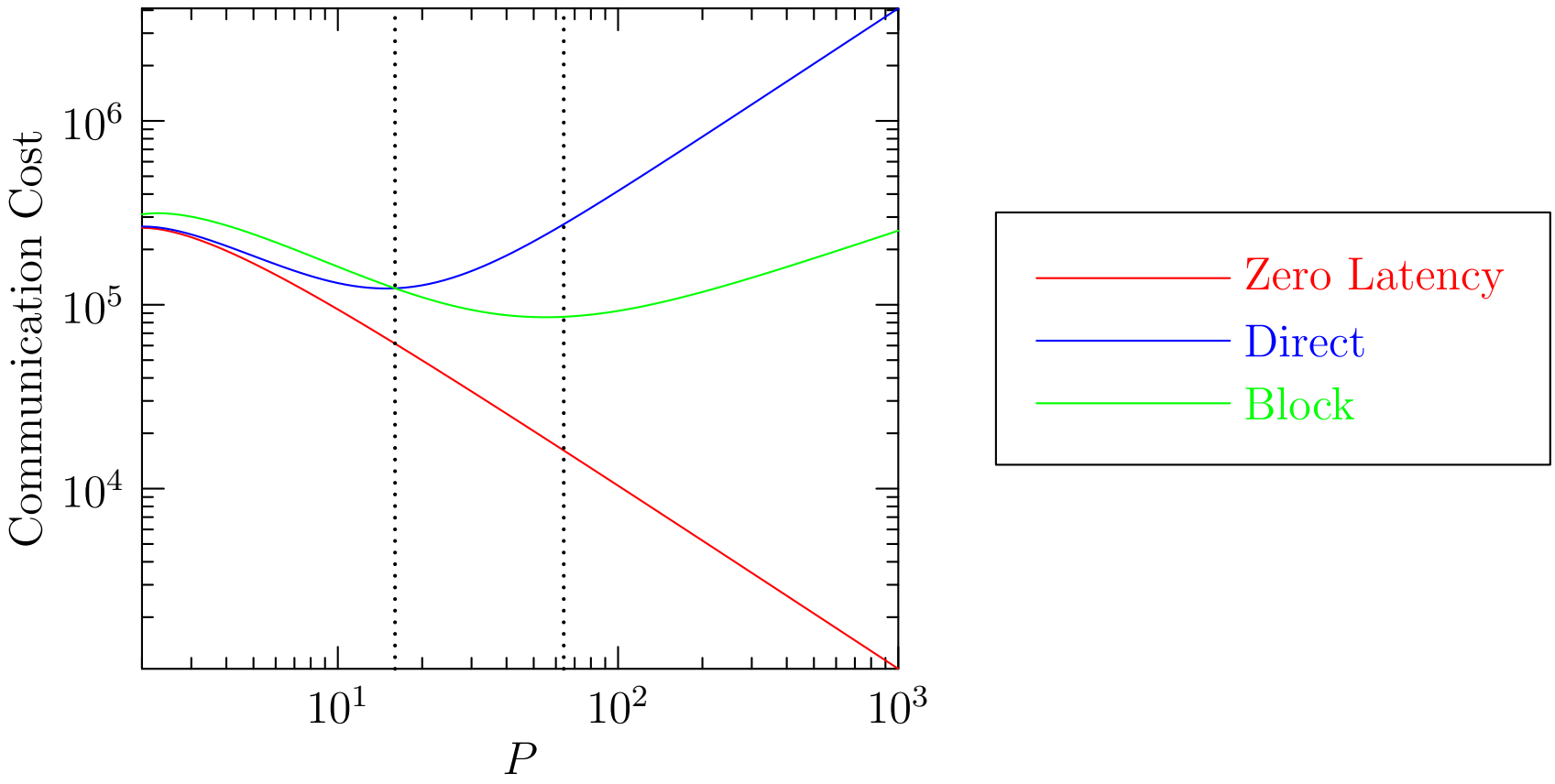
The time required to do a direct transpose is

$$T_D = \tau_\ell (P-1) + \tau_d \frac{P-1}{P^2} m^2 = (P-1)\left(\tau_\ell + \tau_d \frac{m^2}{P^2}\right) \tag{10}$$

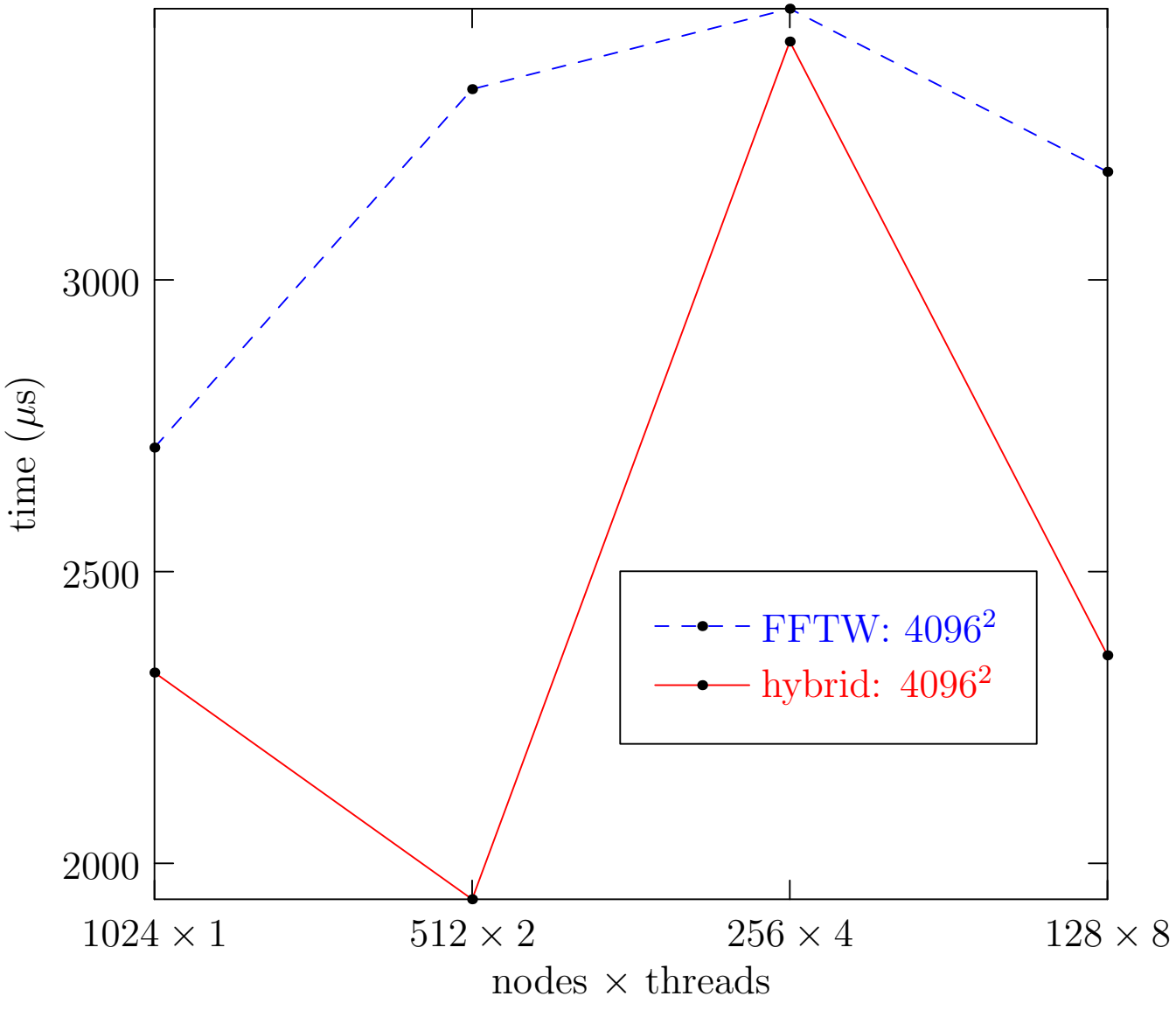
The time for a block transpose is

$$T_B(a) = \tau_\ell \left(a + \frac{P}{a} - 2\right) + \tau_d \left(2P - a - \frac{P}{a}\right) \frac{m^2}{P^2}. \tag{11}$$

# Hybrid Transpose



# Hybrid Transpose



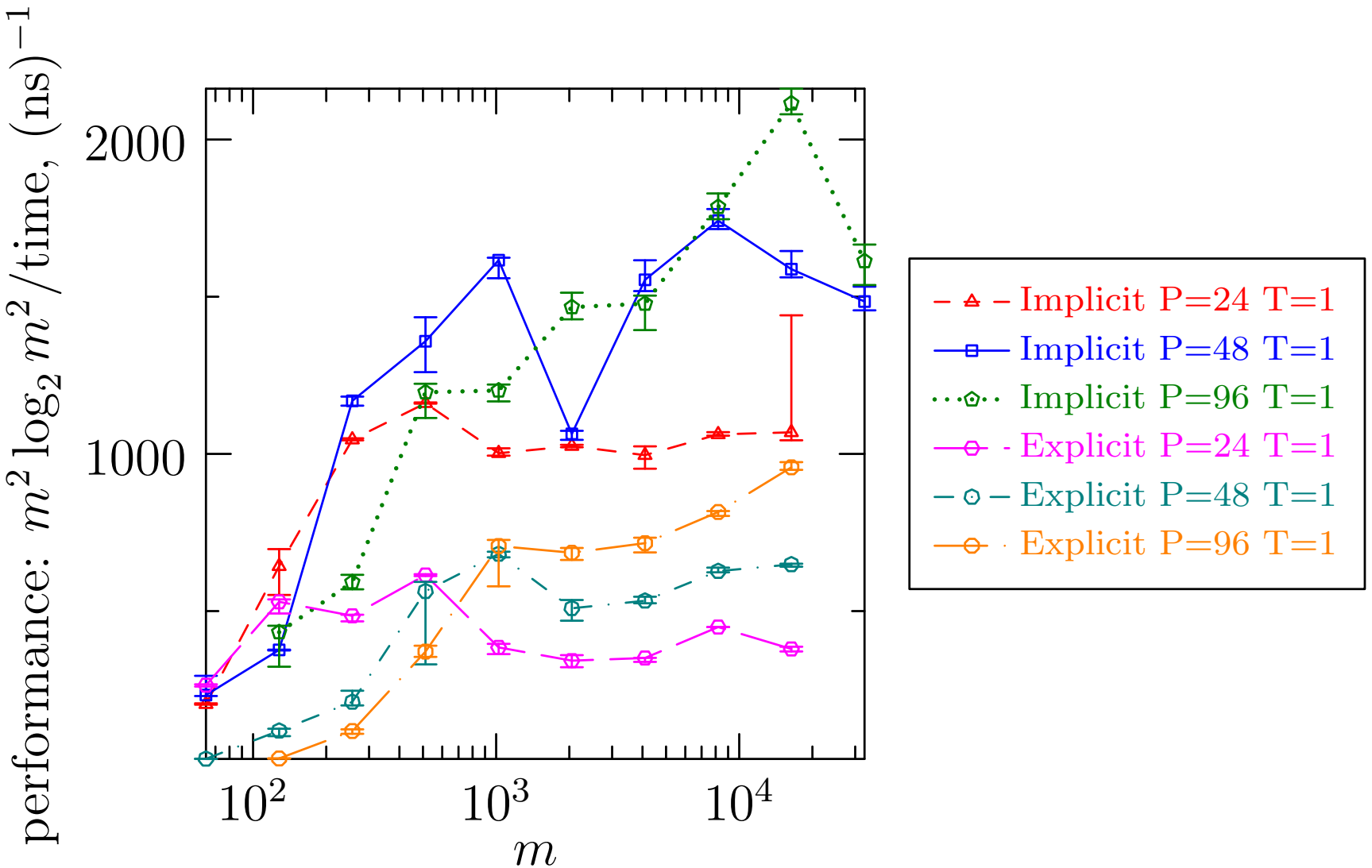
# Hybrid Transpose

The hybrid transpose

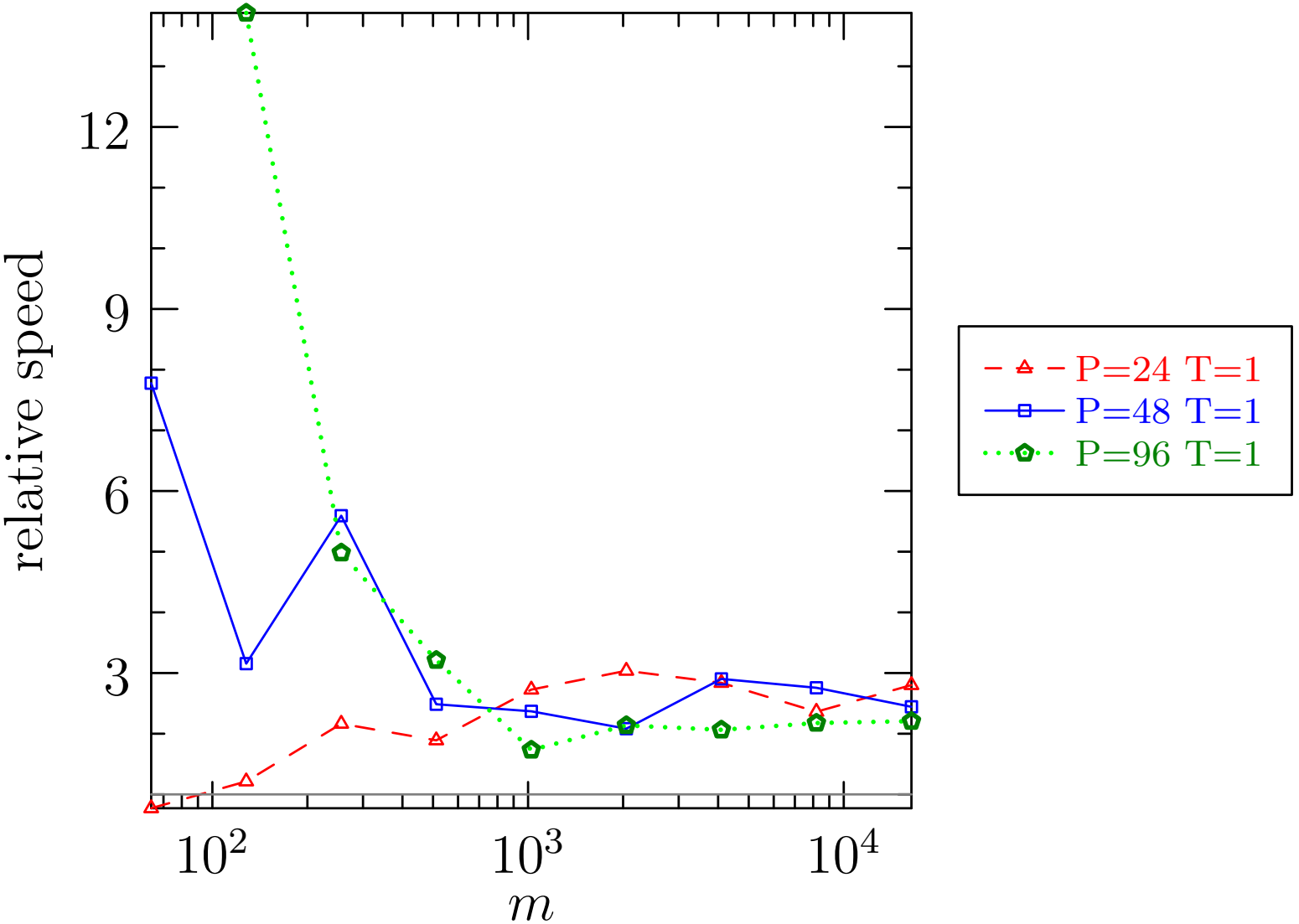
- Uses a direct transpose for large message sizes.
- Uses a block transpose for small message sizes.
- Offers a performance advantage when $P \approx m$.
- Can be tuned based upon the values of $\tau_\ell$ and $\tau_d$ for the cluster.

We use the hybrid transpose in for computing convolutions using implicit dealiasing on clusters.

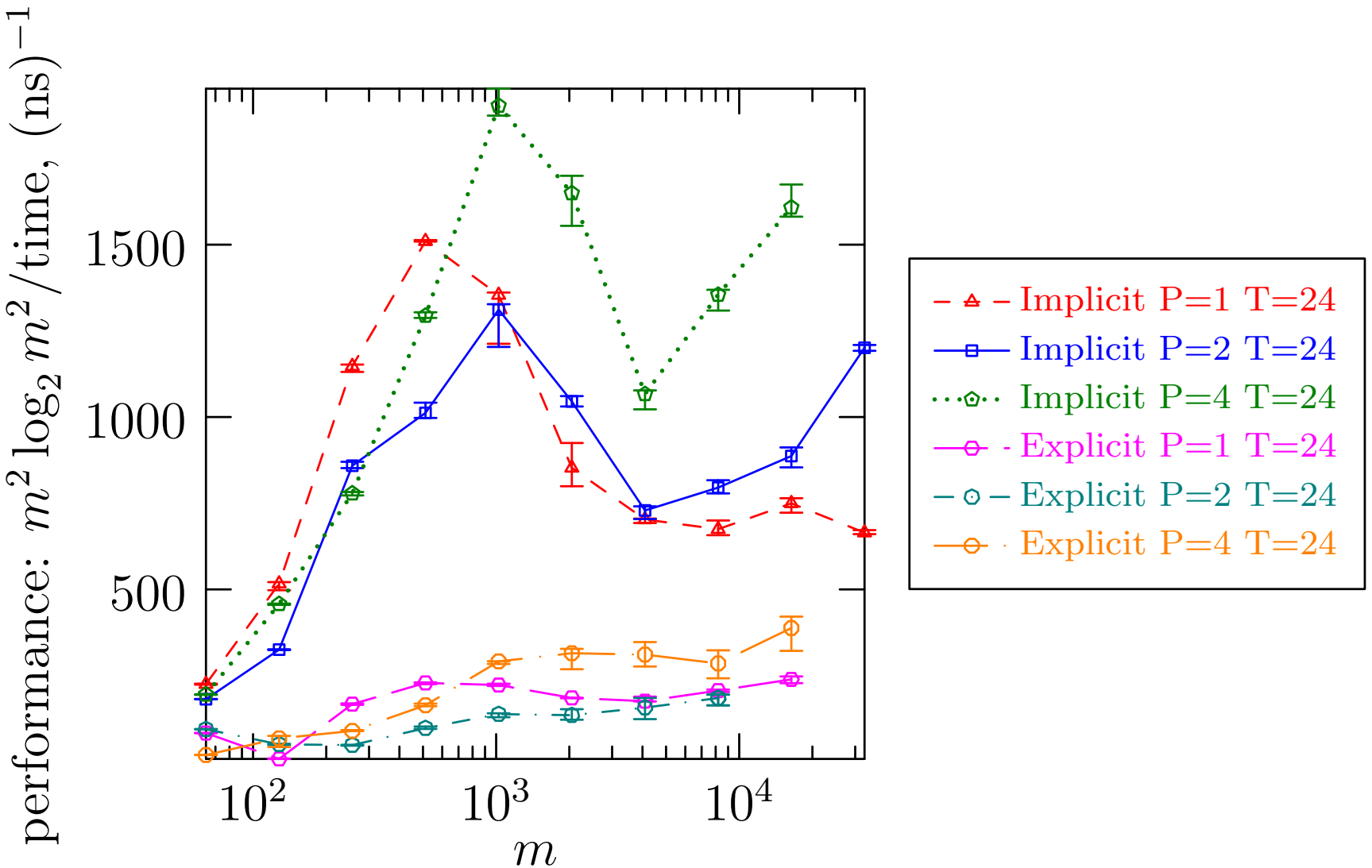
# MPI Convolution: 2D performance



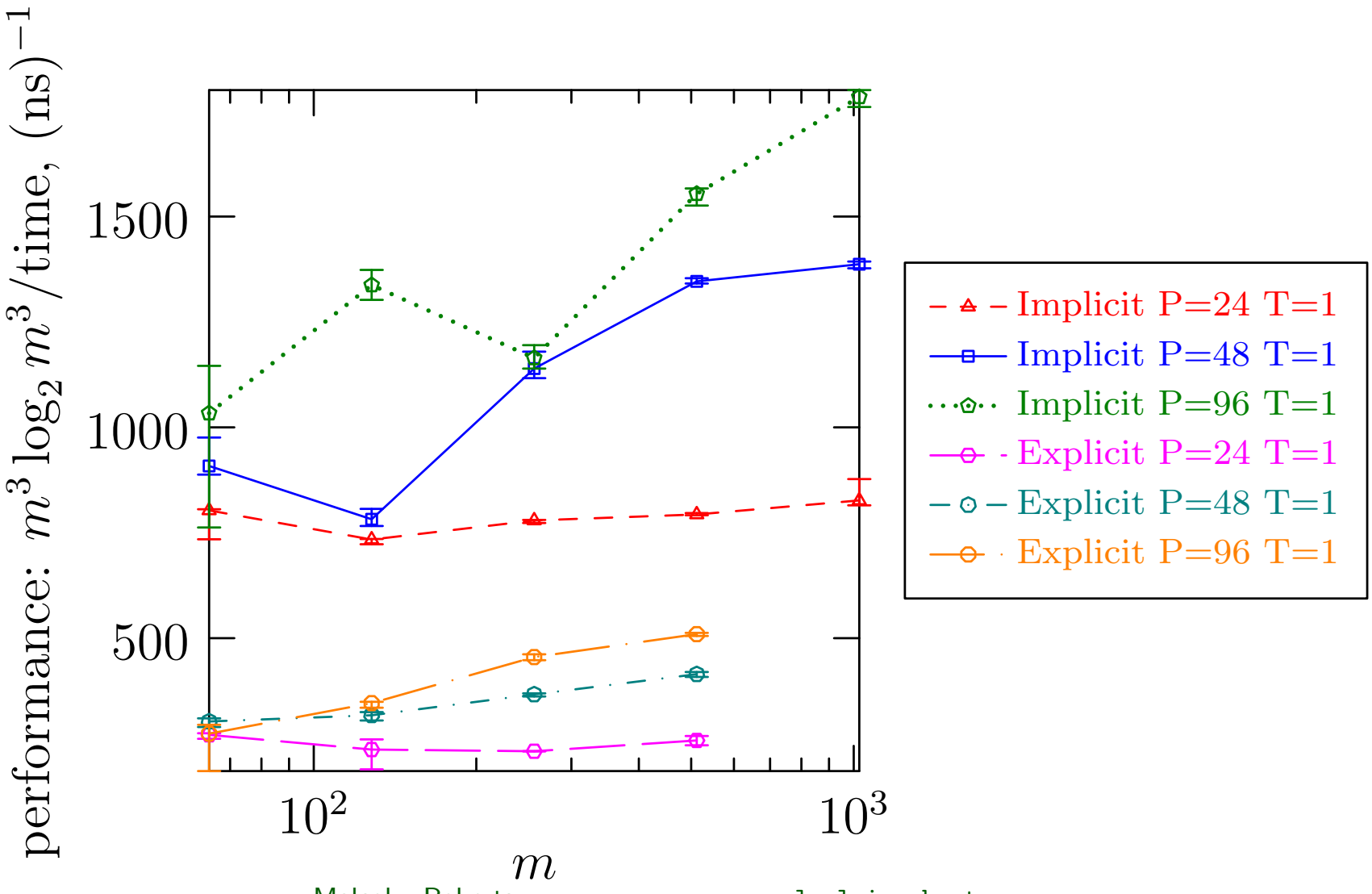
# MPI Convolution: 2D performance



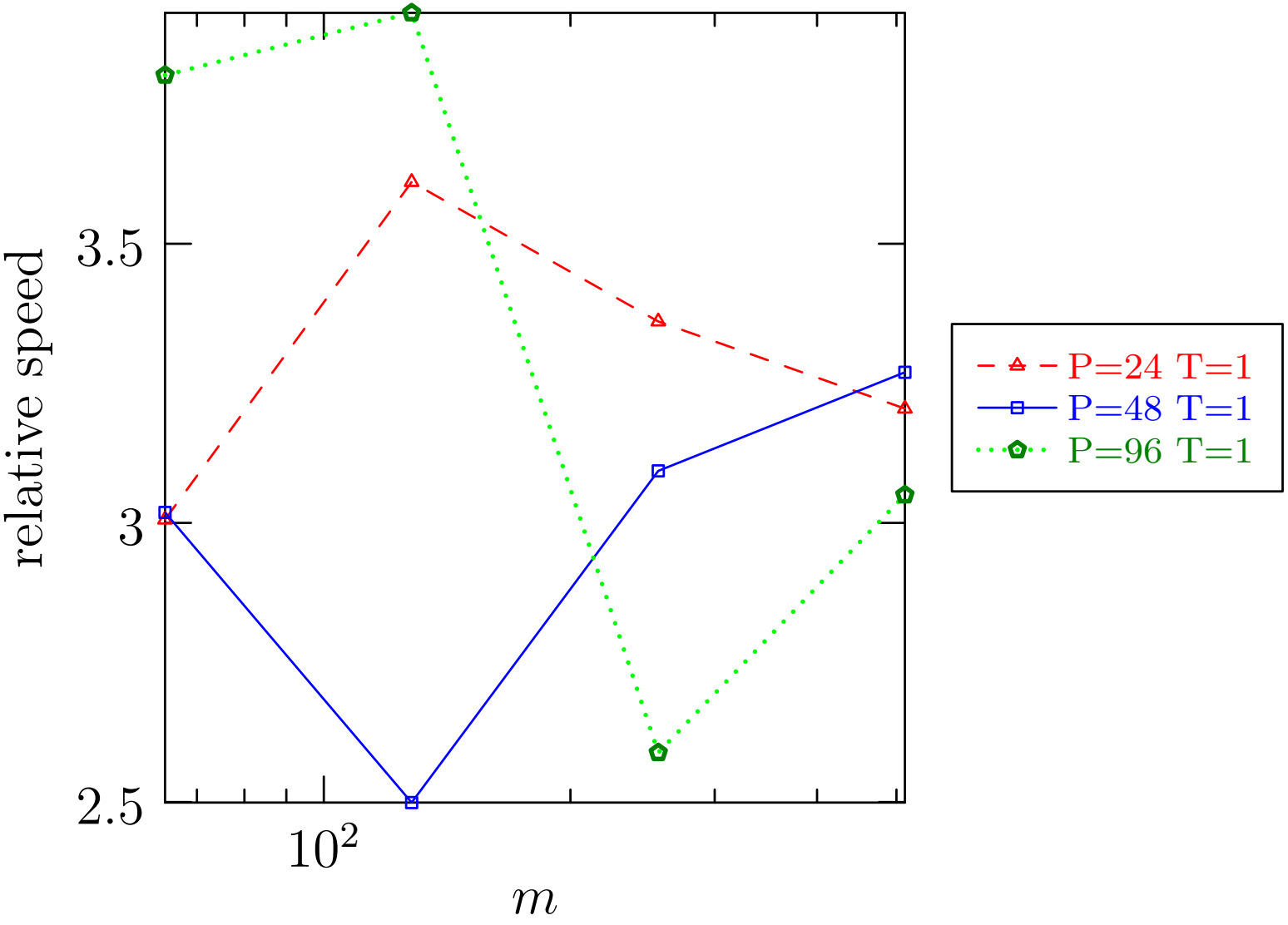
# MPI Convolution: multithreaded 2D performance



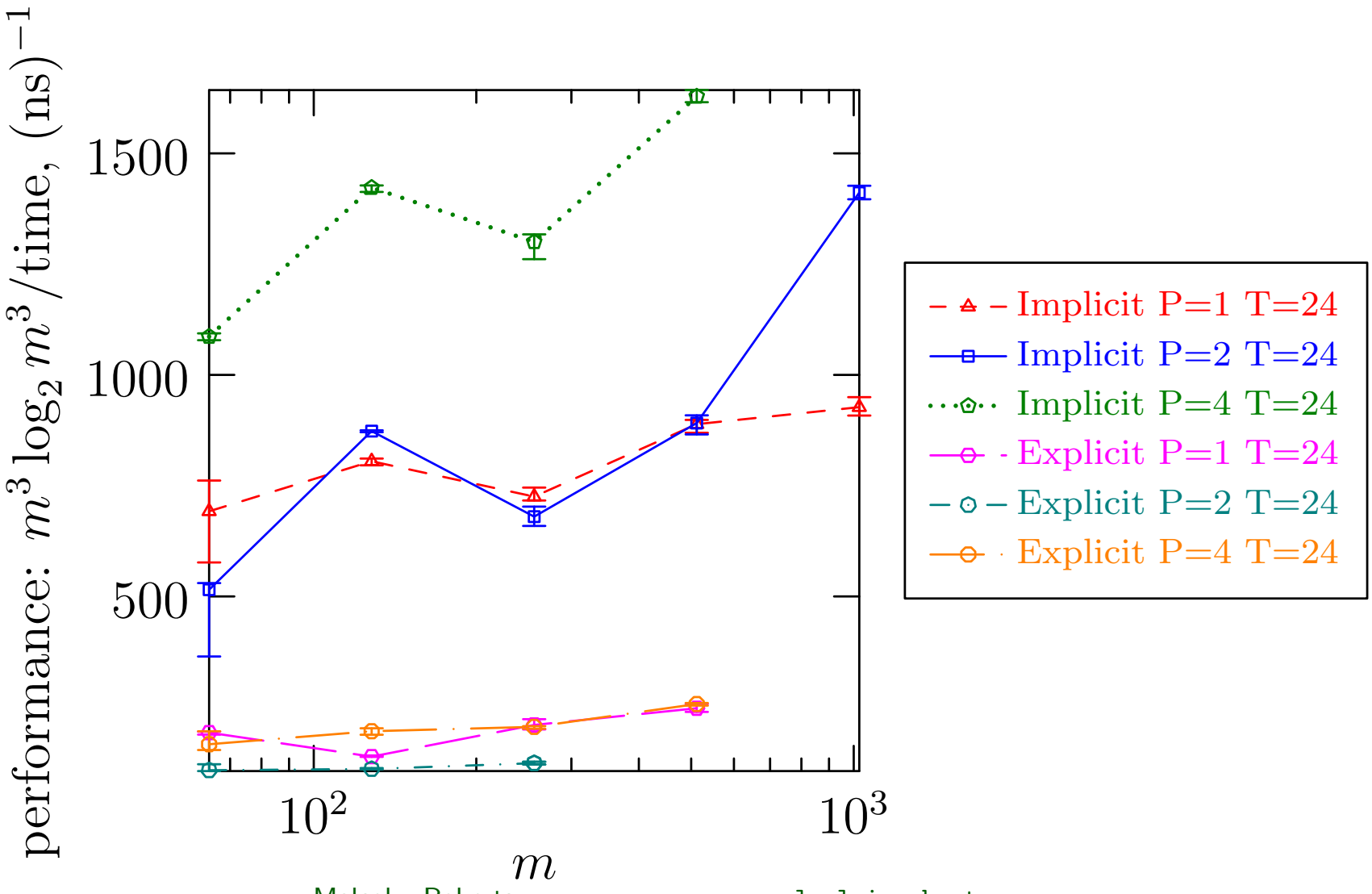
# MPI Convolution: 3D performance



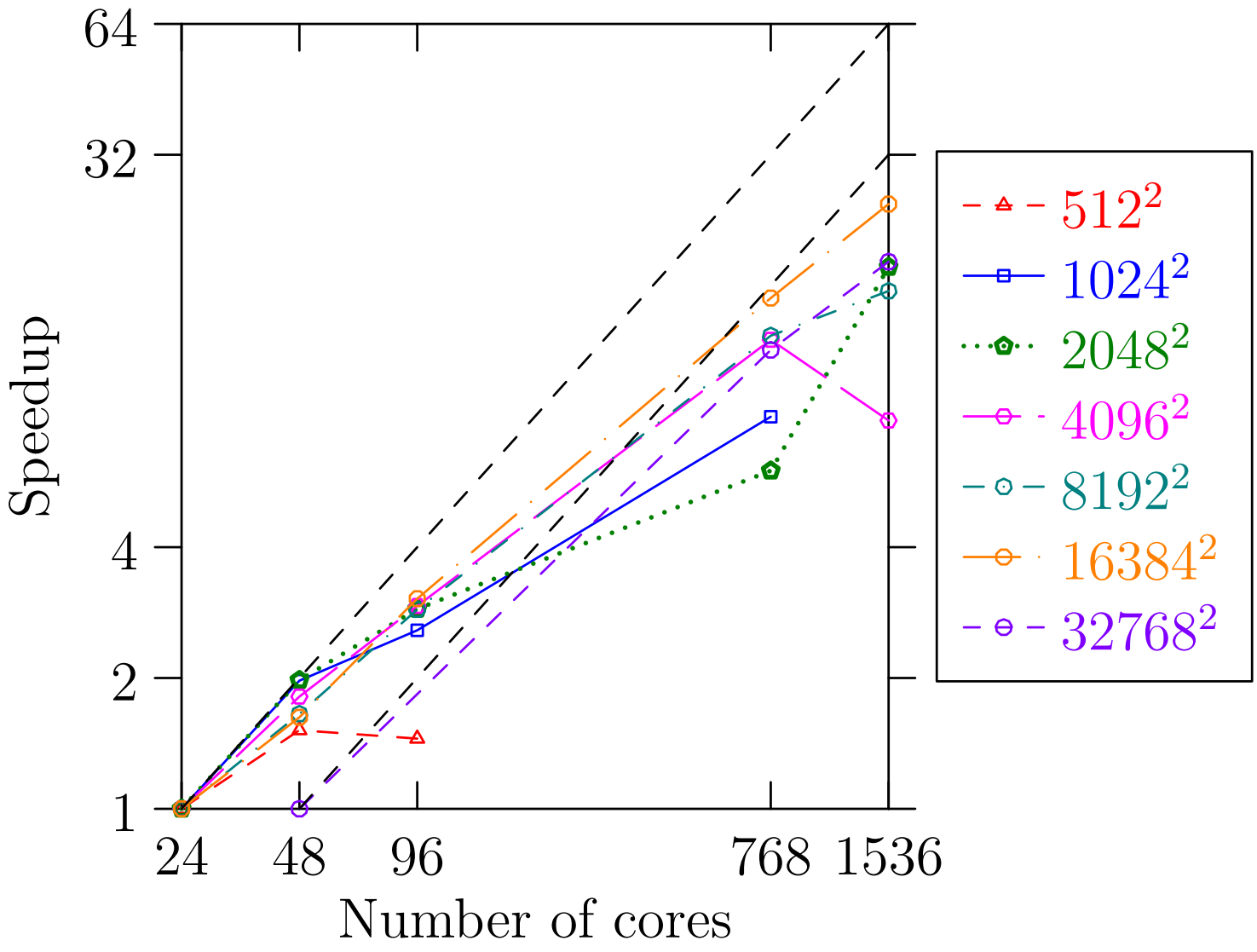
# MPI Convolution: 3D performance



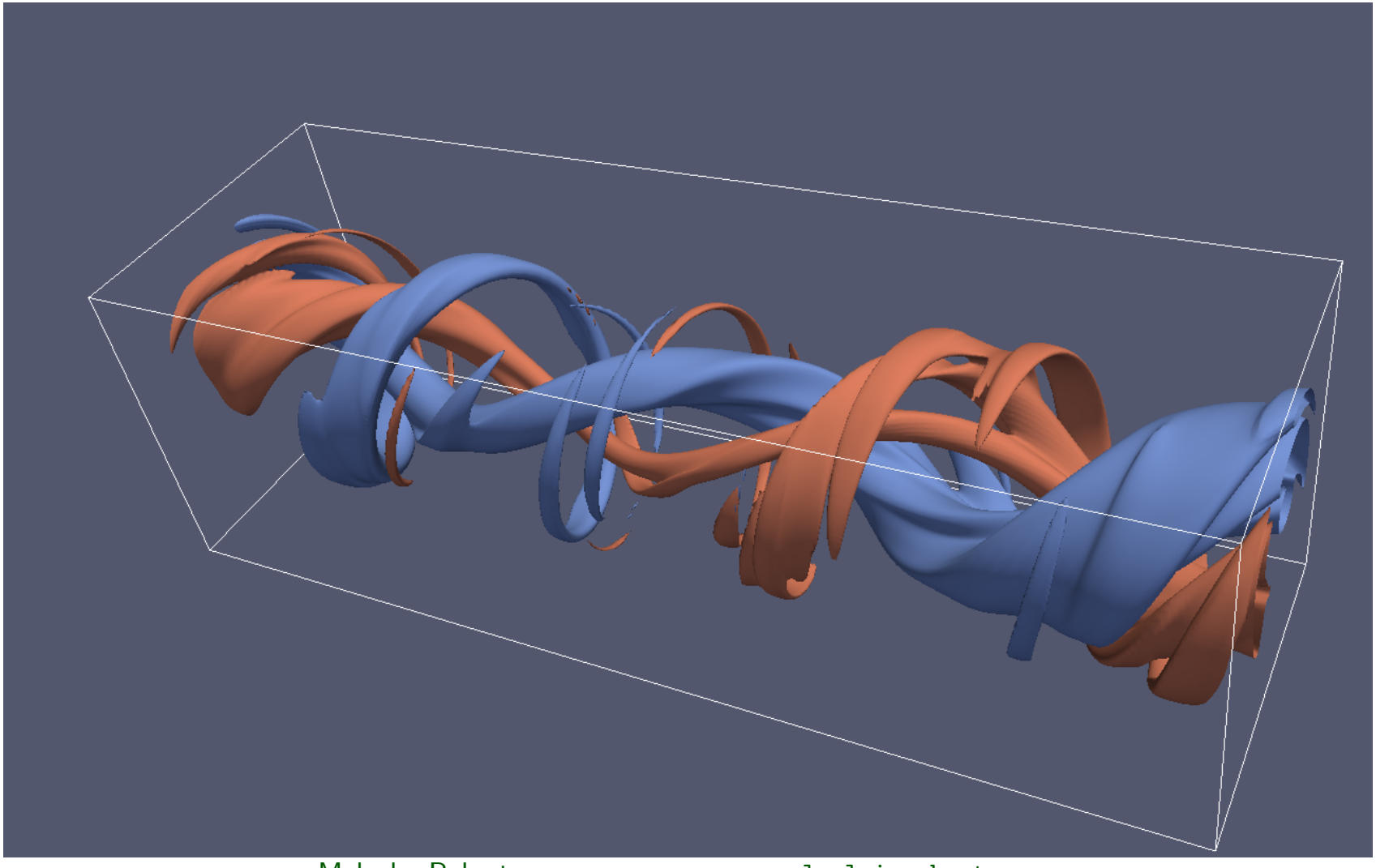
# MPI Convolution: multithreaded 3D performance



# MPI Convolution: 3D scaling



# Application of convolutions: Pseudospectral simulation



# Convolutions Summary

Implicitly dealiased convolutions:

- use less memory
- have less communication costs,
- and are faster than conventional zero-padding techniques.

The hybrid transpose is faster for small message size.

Collaboration with John Bowman, University of Alberta.

Implementation in the open-source project `FFTW++`:

`fftwpp.sf.net`

We have around 13 000 downloads (plus clones).

# Running on GPUs

Computing on general-purpose GPU has two advantages:

- High performance
- Low energy consumption

There are a variety of options for running on GPU:

- CUDA: Libraries available, tools available. Nvidia-only.
- OpenMP 4.0: pragma-based, high-level.
- OpenACC: Being rolled into OpenMP
- OpenCL: Similar to CUDA, but released later.
  - Works on all vendors, very flexible.
  - Runs on GPUs, CPUs, mics (Xeon Phi).

# OpenCL

One writes a normal program, in which the code for the GPU is contained in a string.
At run-time, the program:

1. Selects the OpenCL platform(s) and device(s).
2. Creates an OpenCL context and queue.
3. Compiles the programs into kernels.
4. Allocates buffers on the device.
5. Launches kernels in the queue: managed with events.

# OpenCL

Kernels are the code from the interior of loops.

Example: the C code

```
void myfunc(double* a, double* b, int n) {
for(int i = 0; i < n; ++i) {
  a[i] *= b[i];
}
```

becomes:

```
kernel void mykernel(__global double* a,
                     __global double* b) {
  int i = get_local_id(0);
  a[i] *= b[i];
}
```

# OpenCL

Since the kernel has no loop dependencies, everything is vectorized.

The `__global` keyword specifies that one uses the global device memory.

One has access to the cache with `__local`; if one wants to have data in the cache, then one writes a loop to put it there.

Coalescent memory access is crucial.

So, one has a lot of control, but there's a bit more work.

But the performance is good!

# OpenCL

We developed a discontinuous-Galerkin code for solving hyperbolic conservation laws:

schnaps

**S**olver for **C**onservative **H**ypebolic **N**on-linear systems **A**pplied to **P**lasma**S**

$$\partial_t w + \sum_{k=1}^{k=d} \partial_k F^k(w) = S \tag{12}$$

# schnaps

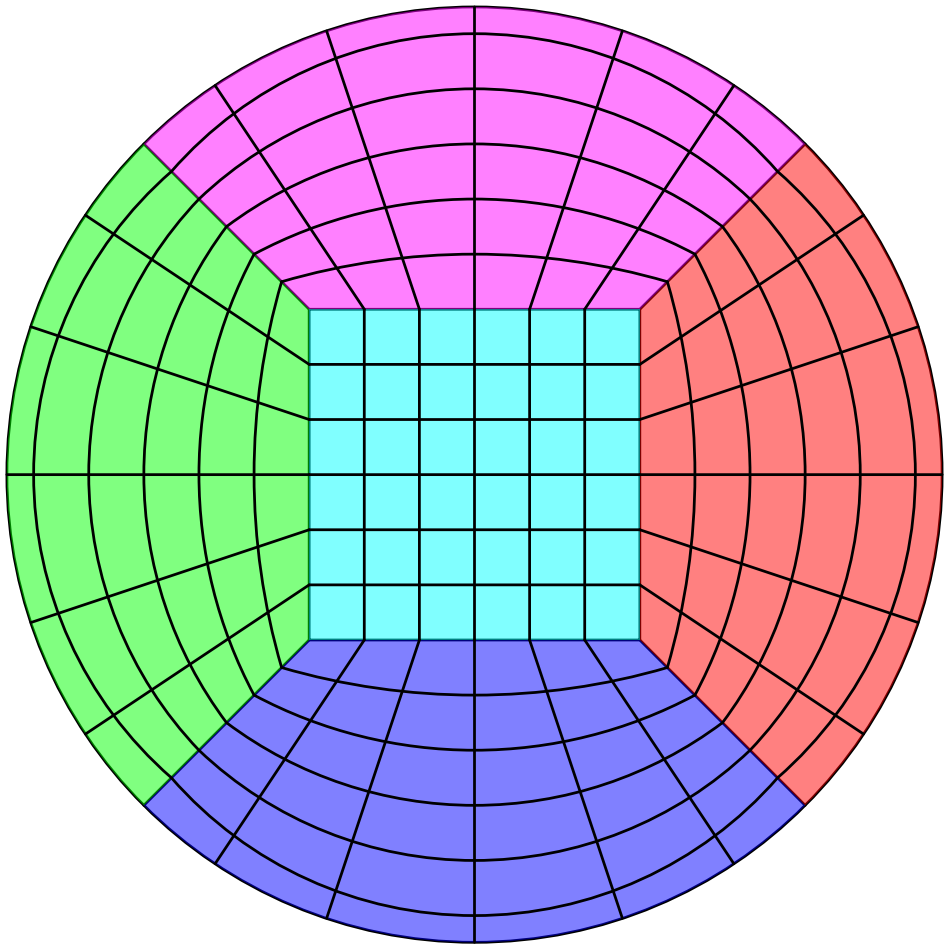
Discontinuous Galerkin method:

- Deals well with complex geometries.
- Local refinement: non-uniform grid.

OpenCL implementation:

- Hexahedral elements for coalescent memory access.
- Macrocell / subcell formulation.
- Array of structs of arrays: yet more coalescence.

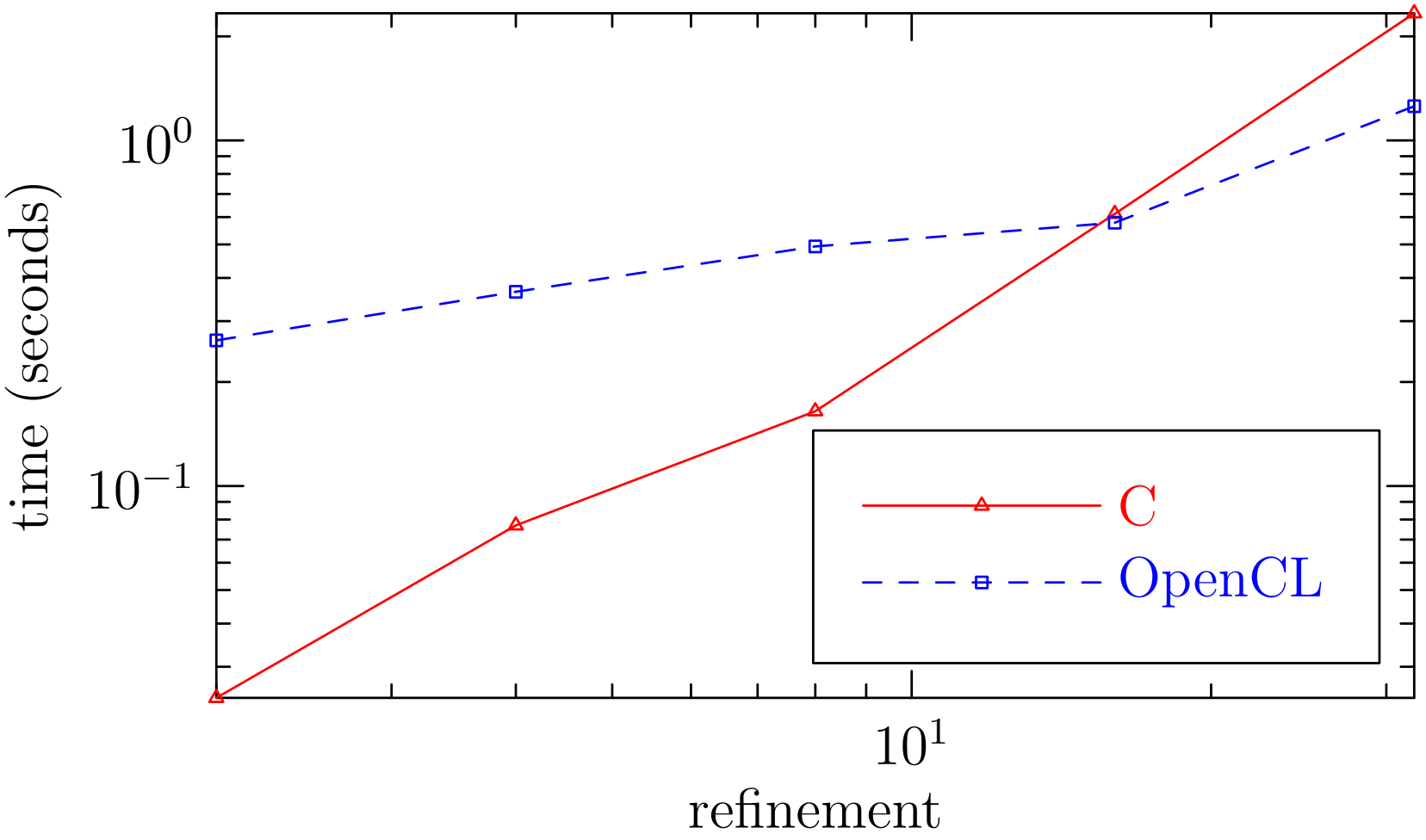
# schnaps



Malcolm Roberts malcolmiwroberts.com

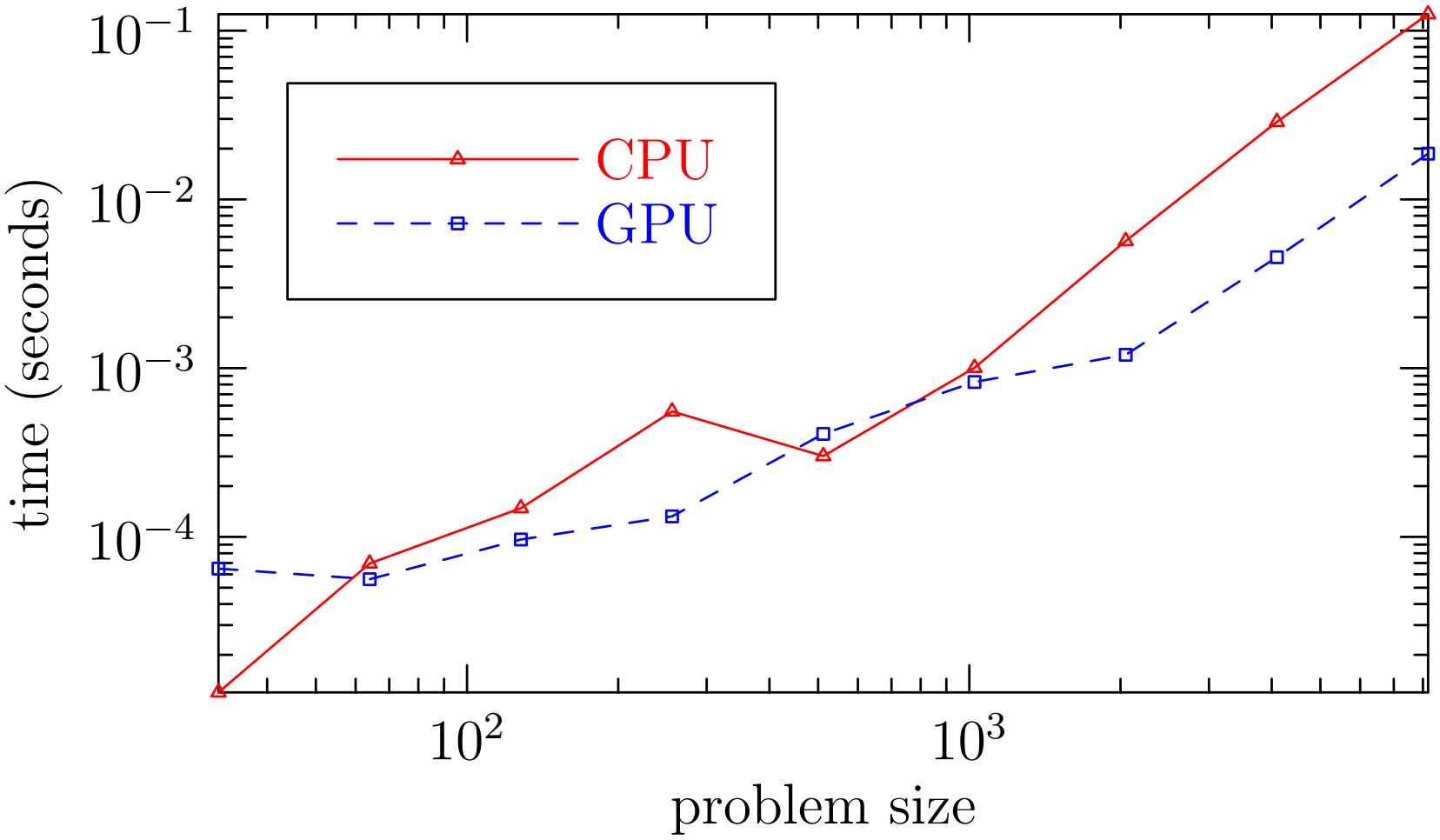
But, is it **fast**?

# Performance analysis of schnaps

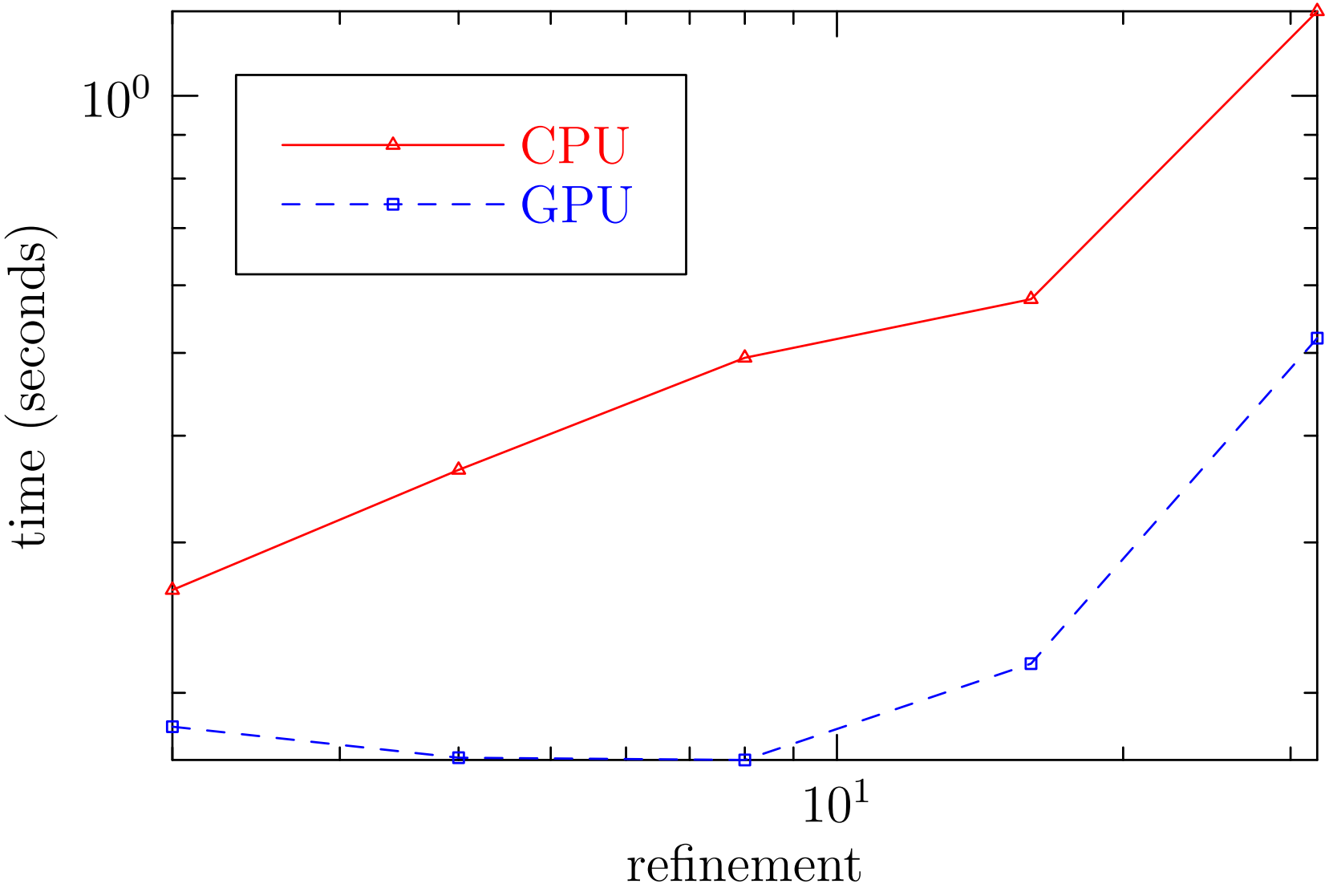


# Performance analysis schnaps

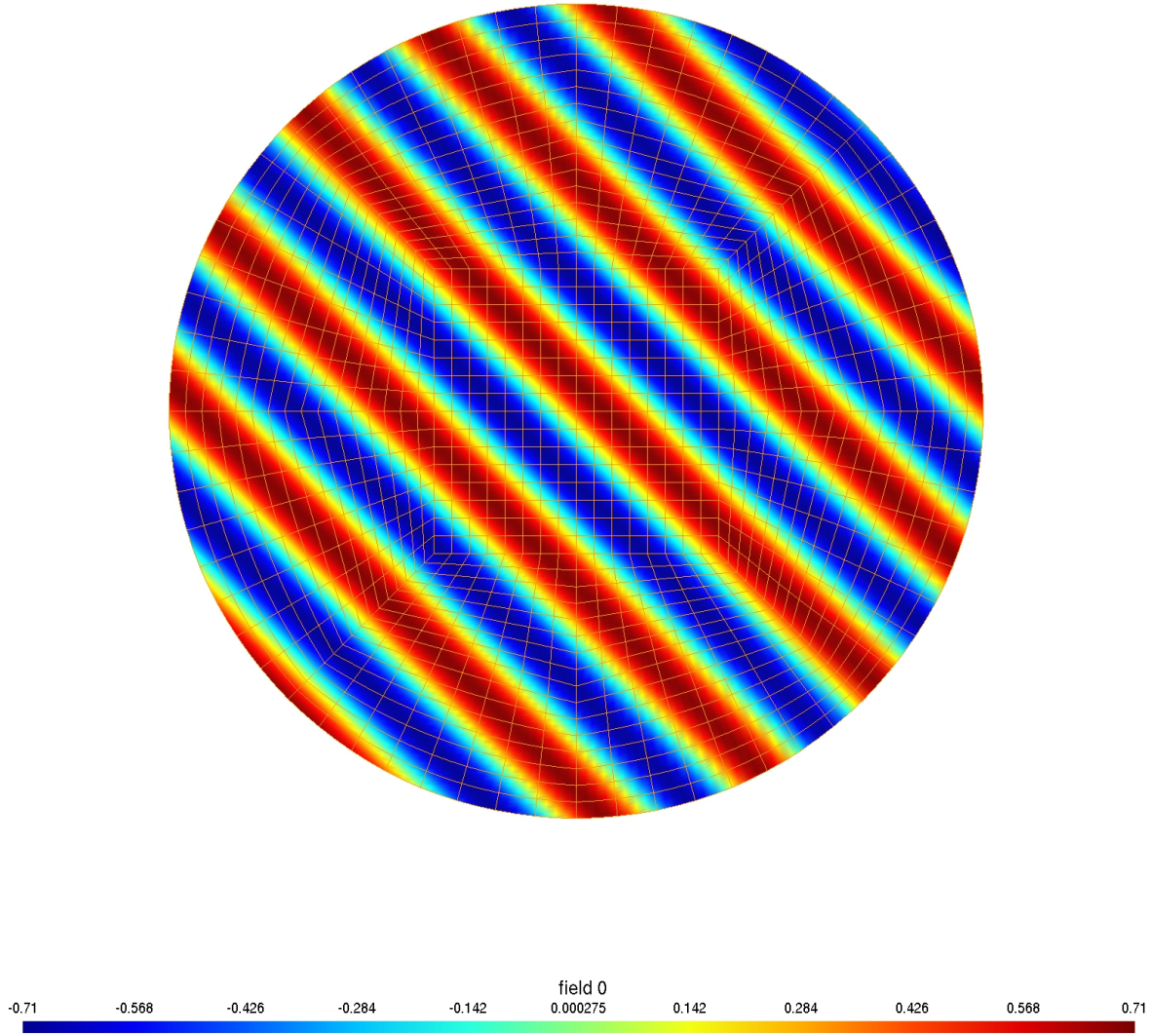
clFFT, an FFT library written in OpenCL by AMD.



# Performance analysis of schnaps



# Example simulation: Maxwell's equations



# schnaps summary

We can conclude that:

1. The C code makes use of all the cores.
2. The C and OpenCL code speeds on the CPU are close for large problem sizes.
3. The performance difference of schnaps between the CPU and GPU is near what we should expect.
4. Thus, we claim that our code makes effective use of the GPU.

We can further improve the code by profiling.

Collaboration with Philippe Helluy and TONUS, University of Strasbourg.

# Conclusion

I presented two projects:

- FFTW++
  - Implicitly Dealiased Convolutions: faster, less memory.
  - OpenMP and/or MPI implementation.
  - Hybrid MPI transpose.
  - Application to a wide variety of situations
- schnaps
  - OpenCL implementation of the discontinuous Galerkin method.
  - Good performance on the CPU, GPU, and mic.

Thank you for your attention!

# Timing statistics