Chameleon: Reactive Task Migration for Hybrid MPI + OpenMP Applications

Jannis Klinkenberg
Chair for High Performance Computing, RWTH Aachen University

Project Partner:

http://www.chameleon-hpc.org/

Chameleon – Project Overview

- **Chameleon**
  - 5^{th} BMBF HPC call
  - Runtime: 01.04.2017 – 31.03.2020

- **Partner**
  - **LMU Munich**
    Chair for Communication Systems and System Programming
    Dr. Karl Fürlinger
  - **RWTH Aachen University**
    Chair for High Performance Computing, IT Center
    Dr. Christian Terboven, Jannis Klinkenberg
  - **TU Munich**
    Department of Informatics
    Prof. Dr. Michael Bader, Philipp Samfaß

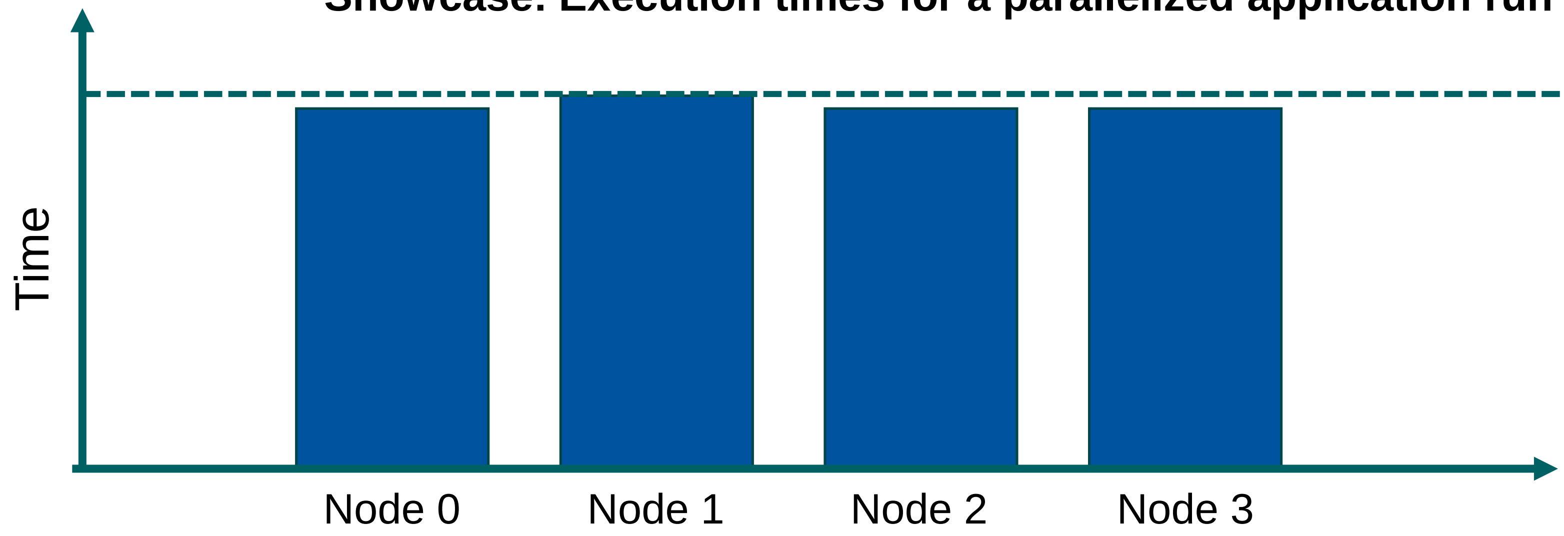
- **Goals**
  - Developing a **task-based** programming environment based and with extensions for **MPI** and **OpenMP** (ease integration into existing applications)
  - Enable applications to react on dynamically changing execution environment

Motivation

Many of todays HPC applications developed with bulk synchronous setup (e.g. MPI + OpenMP)

- Very efficient for bulk synchronous solutions
  - static partitioned domains
  - homogeneous environment

Showcase: Execution times for a parallelized application run



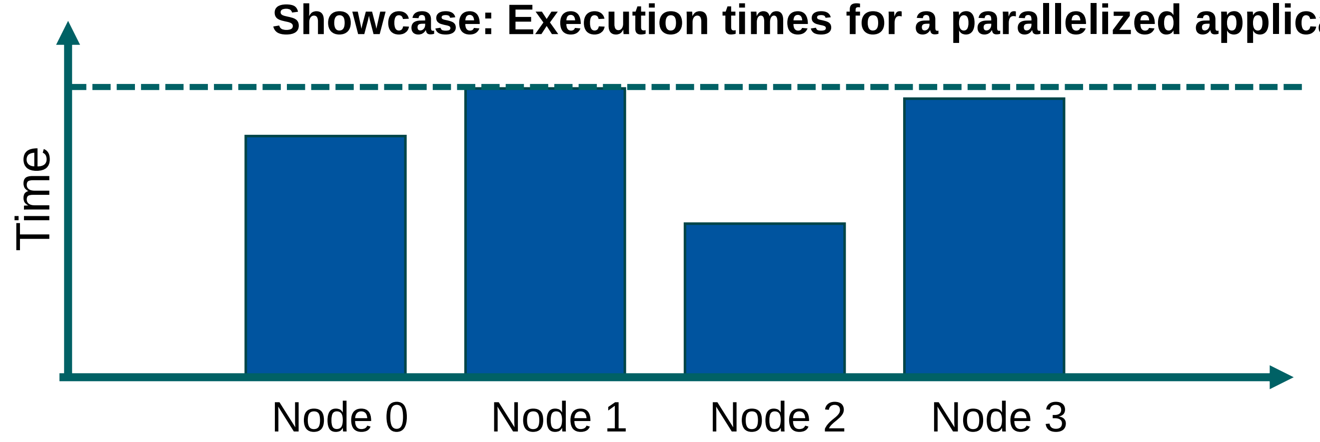
Motivation

Is about to change for current and future HPC systems

Dynamic Variability

- Increasing heterogeneity of systems
  - Complex memory hierarchies (HBM, non volatile memory, DRAM, …)
  - Heterogeneous compute units
- Dynamic adjustment and control based on thermal conditions, …
  - Might affect performance
  - Example: Turbo-Boost mode of modern CPUs

Showcase: Execution times for a parallelized application run



Motivation

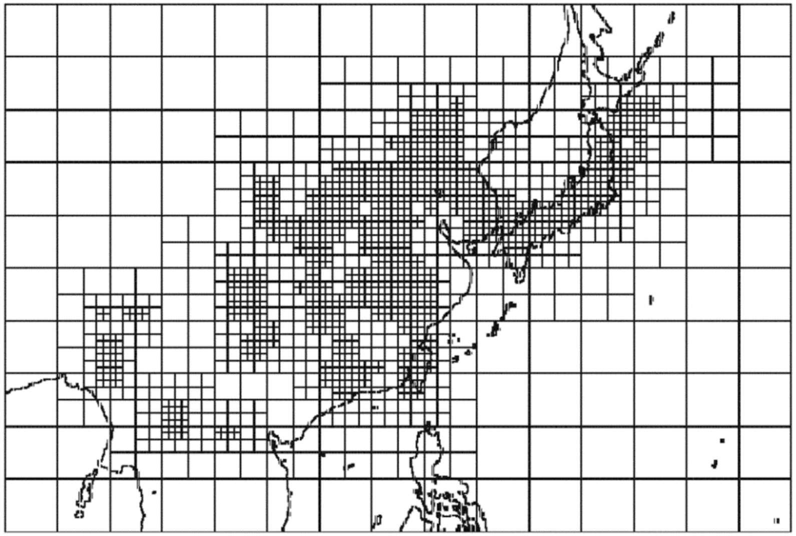
Dynamic variability caused by application

- Example: Iterative algorithms with adaptive mesh refinement (AMR), particle simulations, where workload changes over time

- Showcase application: **sam(oa)²**
  - Finite-Element and Finite-Volume simulations of dynamic adaptive meshes
  - Space Filling Curves (SFC) and Adaptive Meshes for Oceanic And Other Applications (Tohoku Tsunami 2011)
  - Developed at TU Munich

- Depending on situation either refinement or coarsening of cell / section

➢ Might result in load imbalance after each iteration (intra and inter node)



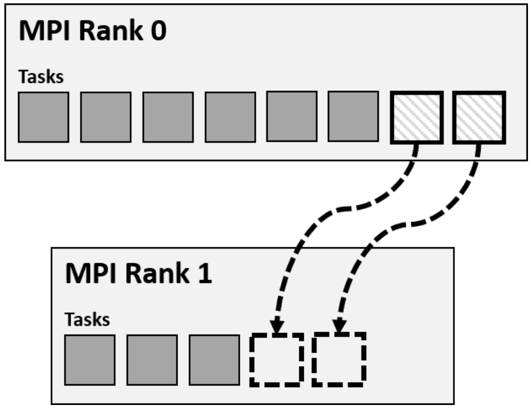
Source: https://doi.org/10.3390/atmos2030484

High Performance Computing IT Center RWTH AACHEN UNIVERSITY

Chameleon Approach: Migratable Tasks + Self Introspection

- **Extend OpenMP tasking / target concept**
  - ➢ Shared memory: Task-to-data affinity (reported in previous talks)
Proposal integrated into OpenMP 5.0 (Chameleon contribution)
  - ➢ Distributed memory: Reactive task migration

- **Migratable task**
  - Basic unit of work without side effects
  - Action + data items (input and/or output)
  - Can be executed locally or migrated to another rank



1. Based on periodically collected introspection data detect imbalance dynamically at runtime

   Result: Rank 0 is significantly slower or has more work

2. Migrate tasks and data to Rank 1

3. Prioritized execution of migrated tasks at Rank 1 + send back results or outputs

➢ Desired: Migrate as soon as possible to overlap communication and computation

Chameleon Approach: Migratable Tasks + Self Introspection

- **Essential Components**

Tasked-based Execution Environment

- ❑ Create, queue and execute migratable tasks
- ❑ Allows early task migration for load balancing between ranks/nodes

Self Introspection

- ❑ Continuous monitoring of the current rank
- ❑ Determine runtime conditions, load or performance metrics

Consolidation and Analysis

- ❑ Consolidates information from all ranks
- ❑ Decision making
  - Migrate tasks?
  - Victim selection

Implementation Objectives

Reactivity
Load imbalances or variability can arise on a very short time scale.
Inevitable to detect these changes as quickly as possible

Smart decision making
Implementation needs to identify emerging imbalances,
decide whether to migrate tasks and select proper victim

Hiding overhead
Migration in distributed memory induces additional overhead. Desired to
migrate tasks as soon as possible to overlap communication and computation

Ease of integration
Augmenting existing applications should not require extensive changes or efforts.
Solution is based on well established standards MPI and OpenMP

Generalization and modularity
General solution applicable to arbitrary applications.
Default behavior with opportunity to customize migration strategy
and incorporate domain or application knowledge

Chameleon – Implementation

CHAMELEON: A task-based programming environment for the development of reactive HPC applications

- **Reactive task migration library written in C/C++**
  - C and Fortran bindings available
  - Based on well established standards MPI and OpenMP

  - Default load specification + migration strategy
  - CHAMELEON Tools Interface
    - Customize / influence load spec. and strategies
    - Incorporate domain / application knowledge

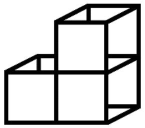
- **Research questions**
  - *Q1:* How do we achieve reactivity and responsiveness?
  - Q2: What is an appropriate general load metric that can be used for arbitrary applications?
  - Q3: When is it recommended to migrate tasks?
  - Q4: How to select proper victims?

Chameleon – Implementation

CHAMELEON: A task-based programming environment for the development of reactive HPC applications

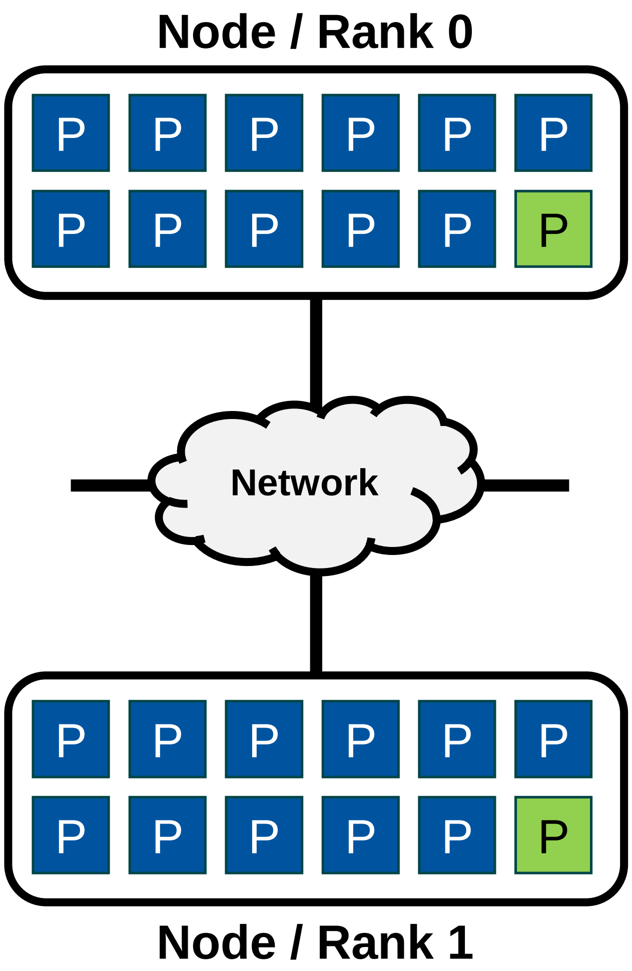
- **Reactive task migration library written in C/C++**
  - C and Fortran bindings available
  - Based on well established standards MPI and OpenMP

  - Default load specification + migration strategy
  - CHAMELEON Tools Interface
    - Customize / influence load spec. and strategies
    - Incorporate domain / application knowledge



- **Research questions**
  - *Q1:* How do we achieve reactivity and responsiveness?
  - Q2: What is an appropriate general load metric that can be used for arbitrary applications?
  - Q3: When is it recommended to migrate tasks?
  - Q4: How to select proper victims?

Implementation – Communication Infrastructure



- **Dedicated communication thread**
  - Bound to last core of CPU set
  - Responsible for continuous actions
    - Introspection (per rank)
    - Communication (load and performance + migration)

✗ One core not available for computation

✓ Guarantees sufficient progression of MPI communication

✓ Essential for fine-granular reactivity and responsiveness

Selecting Proper Migration Victims

- **Initial idea: Migrate to rank with smallest load**
  - Might not be the best choice
  - Can also result in imbalances and overhead

Load	3	4	0	3	1	3
Rank	0	1	2	3	4	5

- 4 ranks with high load
- Idle rank represents minimum
- **After migration:** Rank 2 with load 4

Selecting Proper Migration Victims

- **Sort-based assignment**
  - Sort data by load & find appropriate counter parts
  - Avoids contention and increases overall throughput

Load	8	2	4	3	0	7	5	3	1
Rank	0	1	2	3	4	5	6	7	8

Sorted by load:

Load	0	1	2	3	3	4	5	7	8
Rank	4	8	1	3	7	2	6	5	0

Evaluation

- **Environment: CLAIX (RWTH Aachen University)**
  - Dual-socket Intel Xeon *Broadwell* E5-2650v4 nodes
    - 24 cores @ 2.2 GHz
    - 105 W TDP
  - Intel Omni-Path interconnect
  - Single rank per node + OpenMP thread pinning
- **Compilation with Intel C/C++ or Fortran Compiler 19.0.1 and Intel MPI 2018.4**

- **Executed versions**
  - Classic hybrid MPI + OpenMP without any inter-node load balancing (24 Threads)
  - Hybrid task migration approach (23 Threads)

HW-induced imbalances

- Synthetic dense MxM benchmark
- Each rank has to solve 2400 matrix multiplications
- Enforced power cap (PC) or frequency changes

SW-induced imbalances

- AMR framework sam(oa)^2
- Variation and Imbalances due to refinement

Results Experiments – HW-induced Imbalances

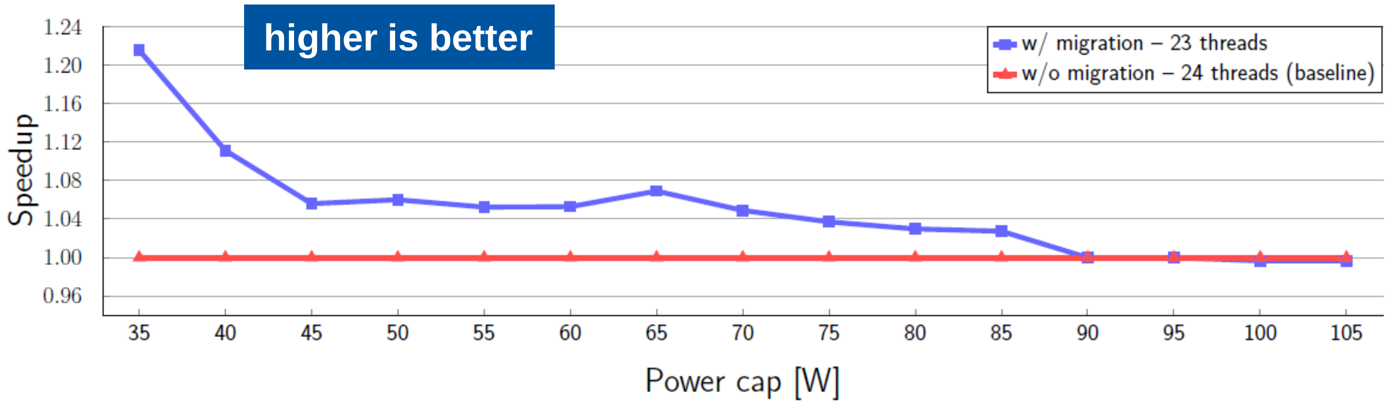


Figure 1: Compensating varying energy efficiency of 4 nodes/ranks under an enforced powercap

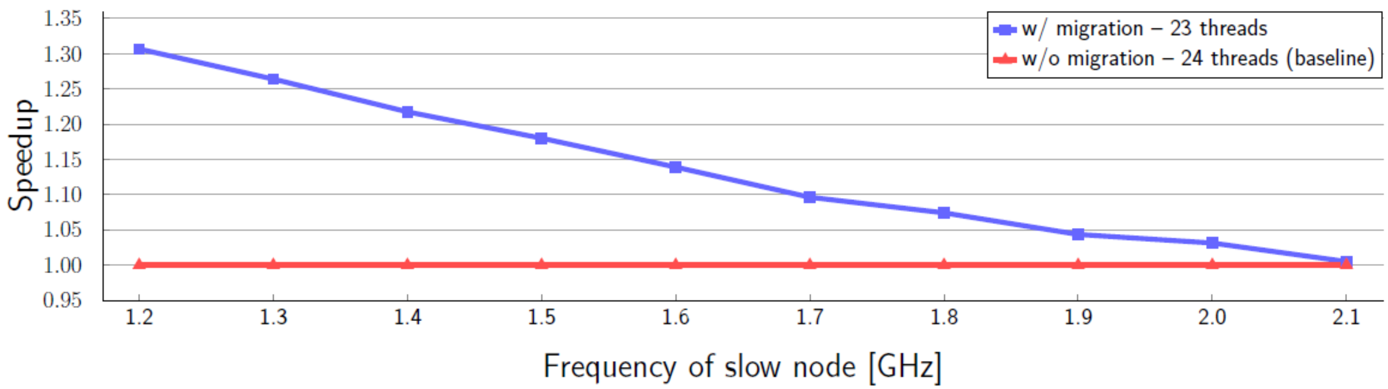


Figure 2: Simulating variations in clock frequency with a single slow rank. Runs have been conducted with 4 nodes/ranks

- With increasing PC varying energy efficiencies visible
- Task migration is able to dynamically balance the load at runtime

Results Experiments – SW-induced Imbalances with sam(oa)²

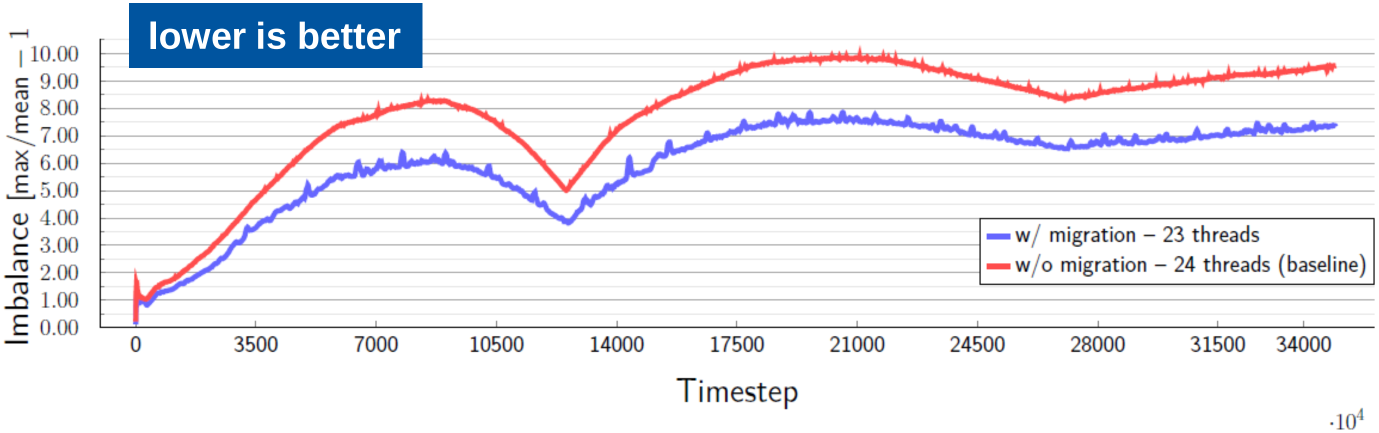


Figure 3: Load imbalances between ranks per time step in sam(oa)² for an application run with 32 nodes/ranks

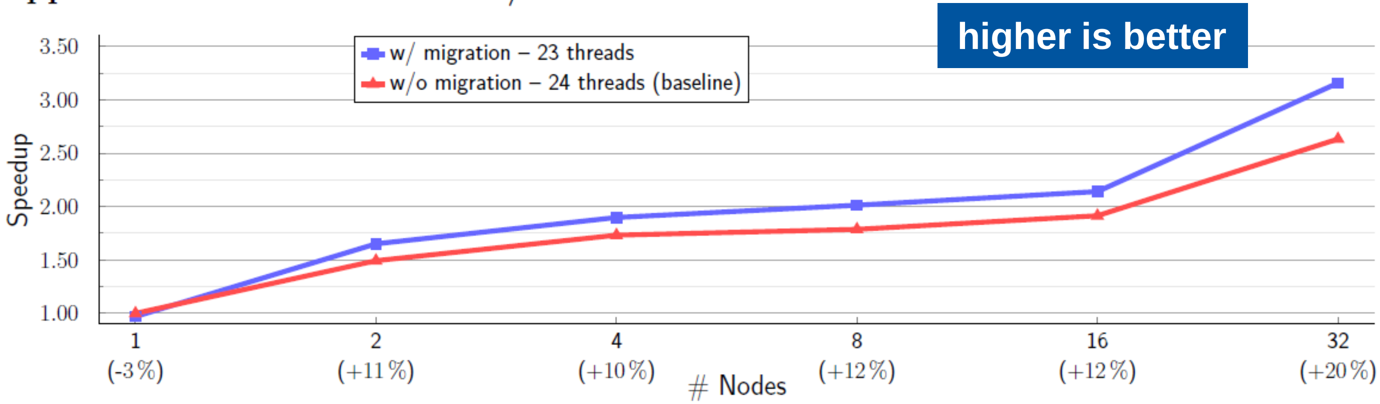


Figure 4: Strong scaling experiments with Tohoku tsunami in 2011 for complete application. Relative speedup to single node base line

- Simulated 60 minutes of Tohoku tsunami in 2011

➢ Reduce degree of imbalance

Summary & Current Topics

Chameleon

- Reactive MPI+OpenMP task migration for fine-granular load balancing
- Robustness against HW- and work-induced imbalances

Code available on GitHub

Current topics

- New reactive concept: Task replication
- Allow dependencies between tasks
- Evaluate different migration strategies, introspection metrics and applications

https://github.com/chameleon-hpc
http://www.chameleon-hpc.org

Thank you!

Backup Slides

Motivation

Ways to tackle load imbalances

- Shared memory
  - Over-decomposition e.g. using OpenMP tasks and task stealing
- Distributed memory
  - Over-decomposition
    - e.g. by using a controller worker pattern to distribute work packages
    - <u>But:</u> Might induce high overhead caused by message and data transfers and requires changing algorithm to new pattern
  - Global repartitioning of data / work
    - Effective predictive technique to ensure proper load balance
    - <u>But:</u> coarse grained, typically exclusive repartition phase and might be too expensive to do that after each iteration
  - Existing frameworks like Charm++, HPX, …
    - High porting effort

➢ **Need a way to dynamically / reactively adapt to changing circumstances**
  - i.e. dynamic load balancing between compute nodes

Code Example (Hybrid MxM multiplications)

```
// function that performs MxM
void compute_matrix_matrix(double *A, double *B, double *C, int mat_size);

int main()
{
  ...
  void* lit_size = *(void**)(&size); // pointer literal representing value of size
  #pragma omp parallel
  {
    #pragma omp for nowait
    for(int i=0; i<num_tasks; i++) {
      double *A = matrices_a[i];
      double *B = matrices_b[i];
      double *C = matrices_c[i];

#if USE_OPENMP_TARGET_CONSTRUCT
      #pragma omp target map(tofrom: C[0:size*size]) map(to: A[0:size*size], B[0:size*size])
      compute_matrix_matrix(A, B, C, size);
#else // API approach
      map_data_entry_t* args = new map_data_entry_t[4];
      args[0] = map_data_entry_create(A, size*size*sizeof(double), MAPTYPE_INPUT);
      args[1] = map_data_entry_create(B, size*size*sizeof(double), MAPTYPE_INPUT);
      args[2] = map_data_entry_create(C, size*size*sizeof(double), MAPTYPE_OUTPUT);
      args[3] = map_data_entry_create(lit_size, sizeof(void*), MAPTYPE_INPUT | MAPTYPE_LITERAL);
      add_task((void *)&compute_matrix_matrix, 4, args);
#endif
    }
    // trigger execution (In background: introspection + task migration)
    distributed_taskwait();
  }
  ...
}
```

Task Entry Point

Task Creation

Task Execution & Termination