❖ A PGAS Framework Supporting a Parallel Computation Expanding and Shrinking its Scale Dynamically ❖

Taura Lab, M2, Kentaro Hara

2010.4.30

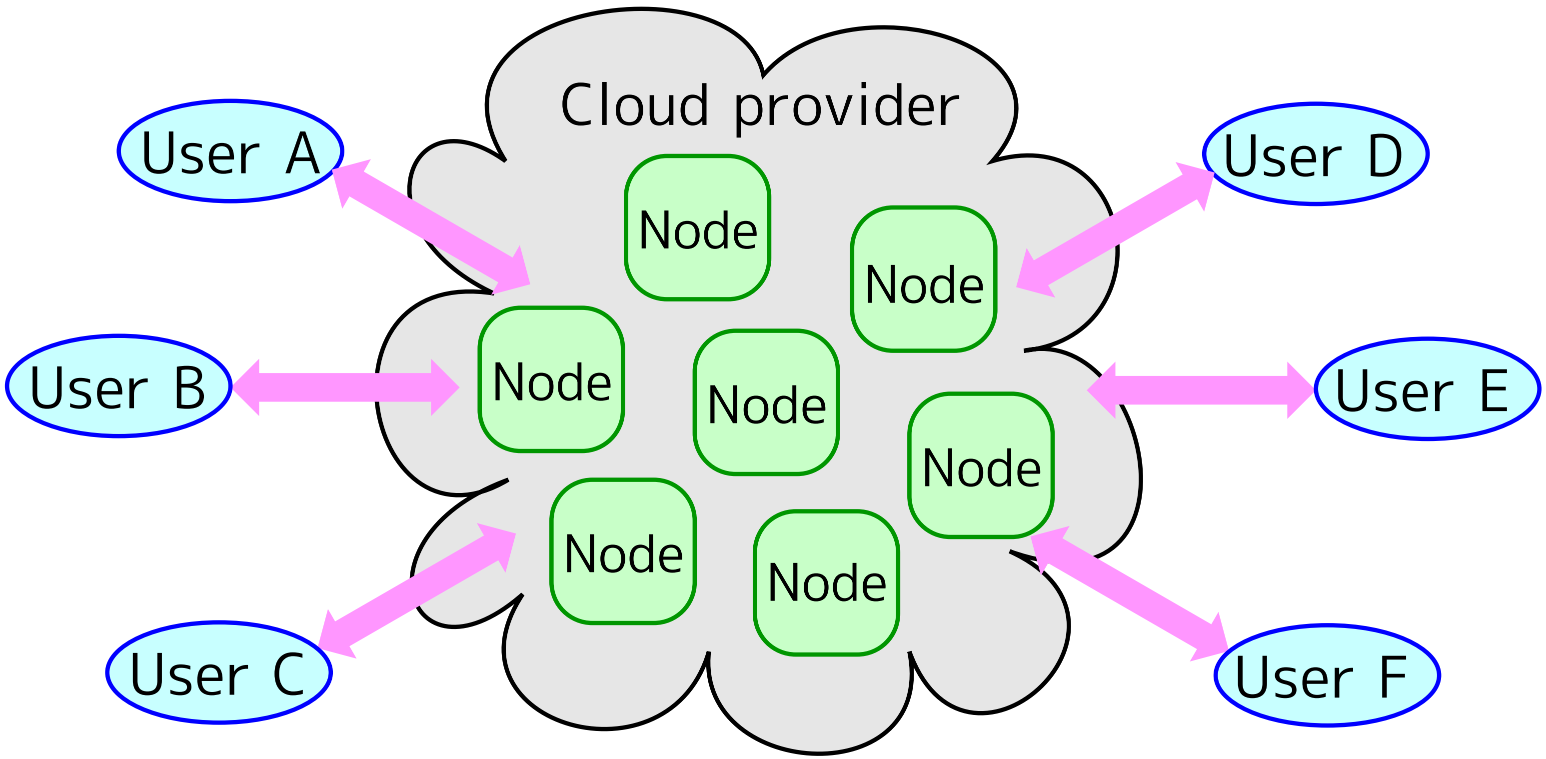
❖ 1. Introduction

Backgrounds and a goal

- Backgrounds: Large-scale parallel scientific computings
  - ➔ Stress analyses
  - ➔ Fluid analyses
  - ➔ Earthquake simulations
  - ➔ ...
- Goal: **Develop a framework which supports these large-scale parallel scientific computings on a cloud**

What is a cloud?

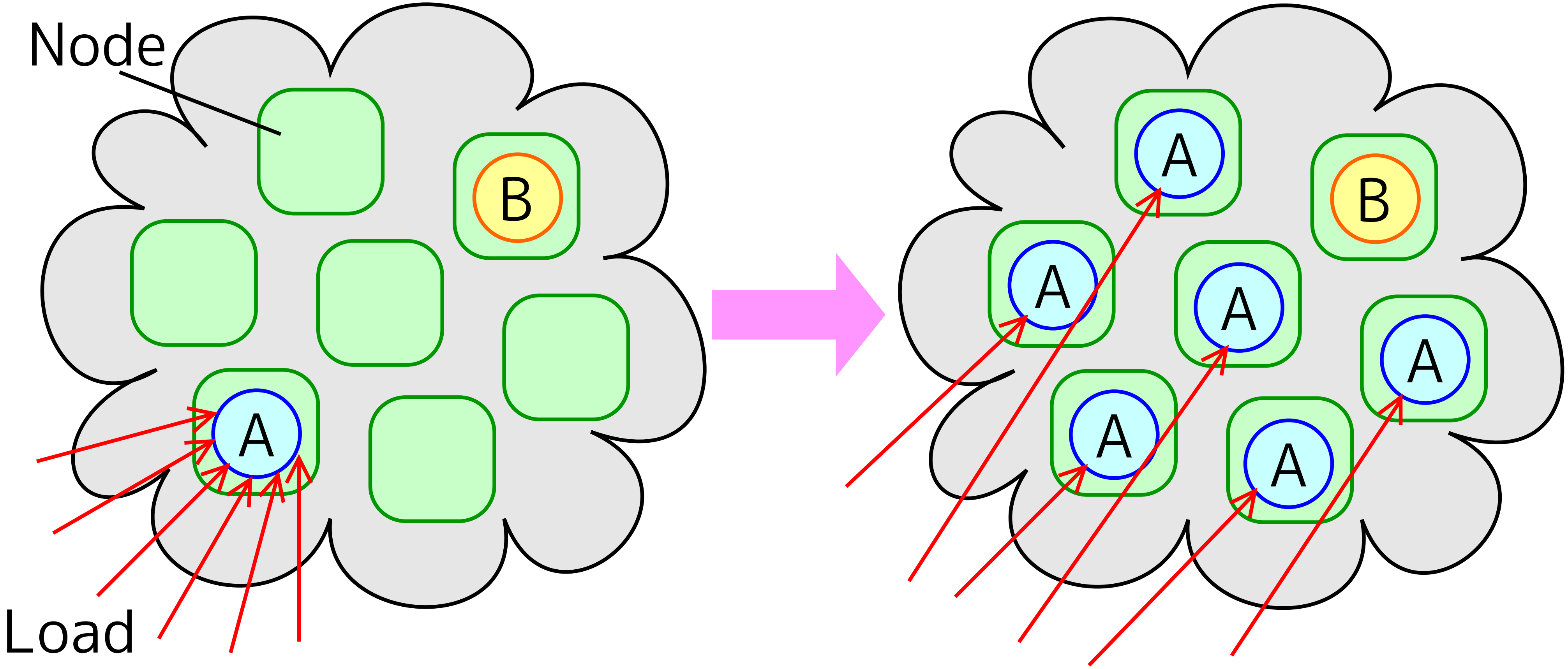
- Mechanism:
  - A provider manages a data center and provides its resources **as a service**
  - A user can use as many resources as needed in a pay-as-you-go system
- Model: **Multiple resources are used by multiple users**



Use as you like "as a service"

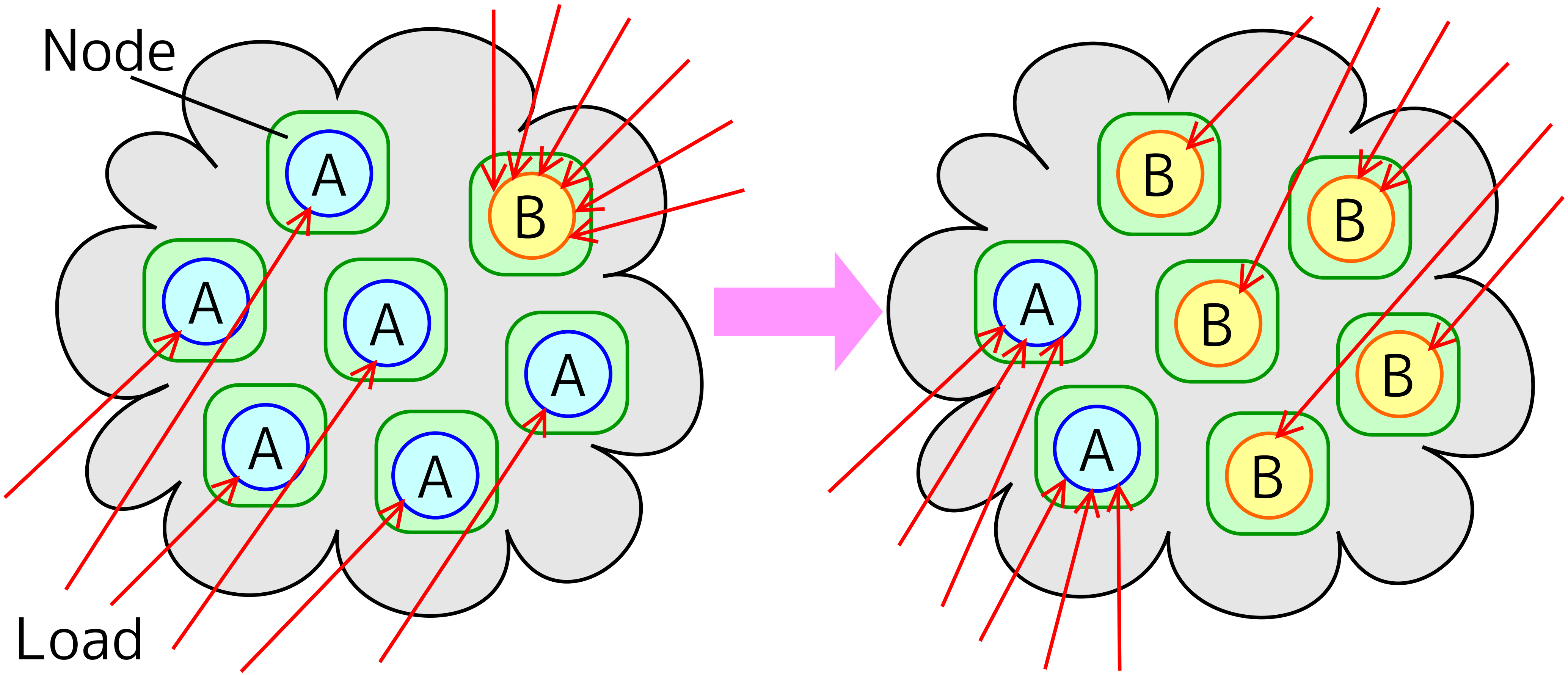
An example of the cloud(1)

- When the load of a user A increases, the computational scale of the user A expands



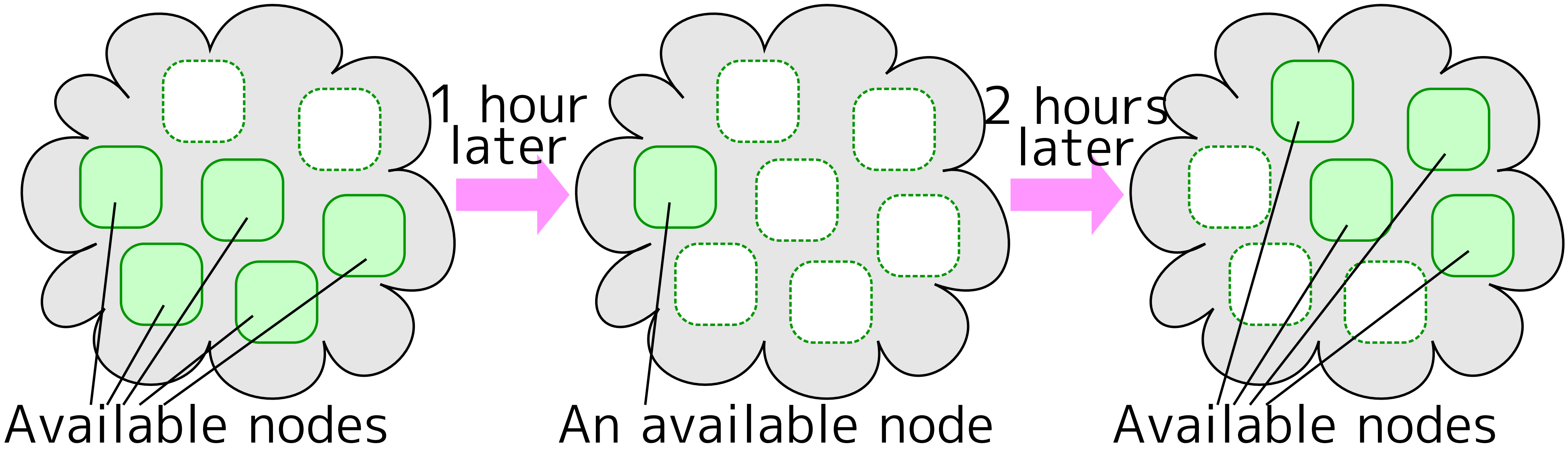
An example of the cloud(2)

- Later, when the load of a user B increases, the computational scale of the user B expands instead of shrinking the scale of the user A



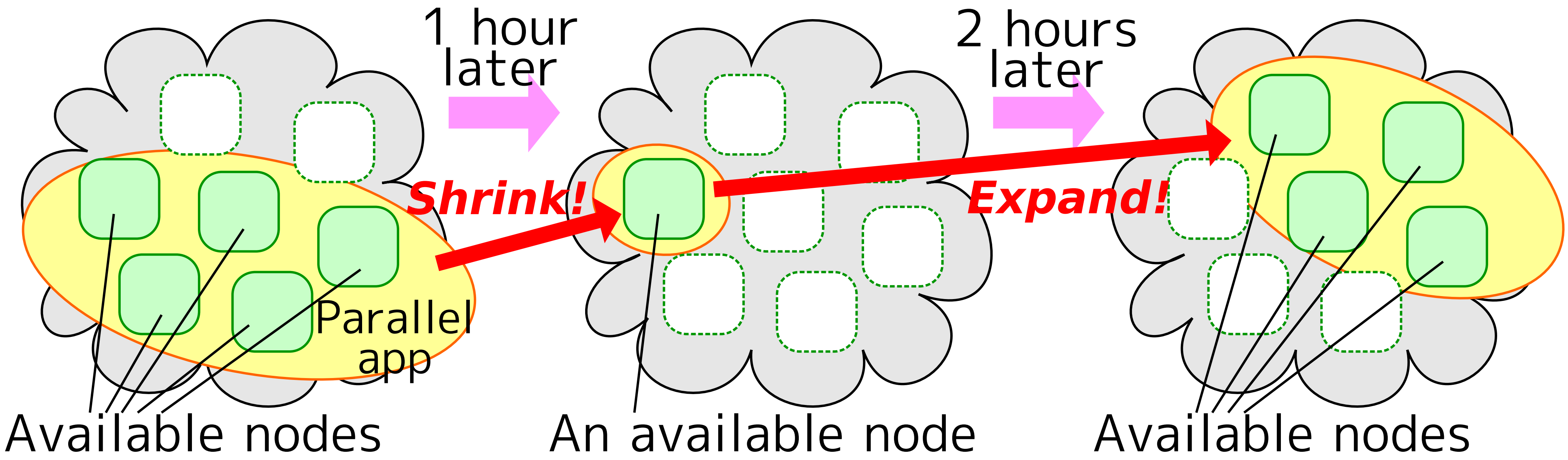
The essence of the cloud

- Multiple resources are used by multiple users
- Hence **available resources for each user change dynamically in response to the overall load**



Then, how should a parallel computation run on the cloud?

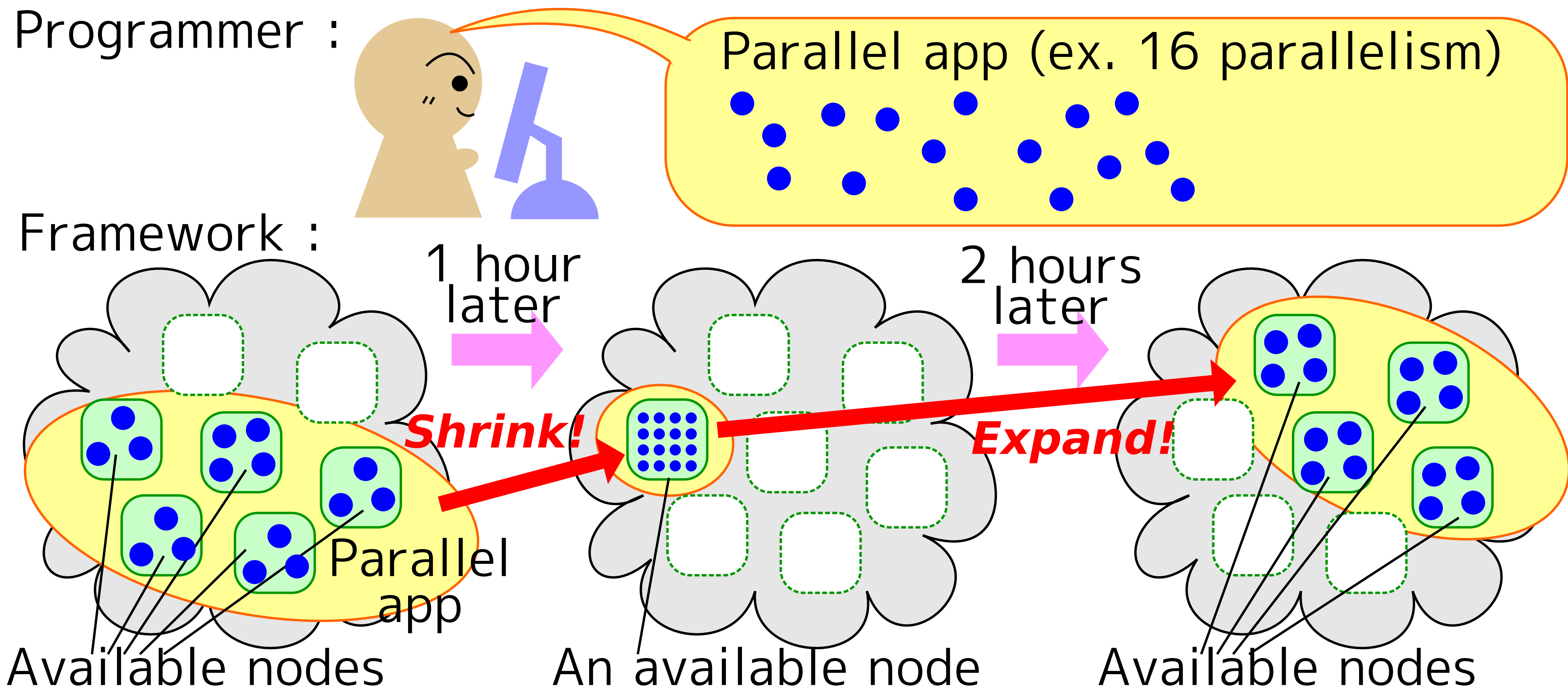
- Targeted apps: **Long-running** large-scale parallel scientific computings
  - ➔ Finite element methods (FEM)
  - ➔ Particle methods
- These apps should run **expanding and shrinking their scale dynamically in response to the available resources at the time**[Chaudhart et al,2006]
  - ➔ "But... too difficult to develop such an elastic parallel program!!!"



A required programming model

(1) A programmer only has to describe the parallelism of an app

(2) Then, a framework expands and shrinks the computational scale automatically and dynamically

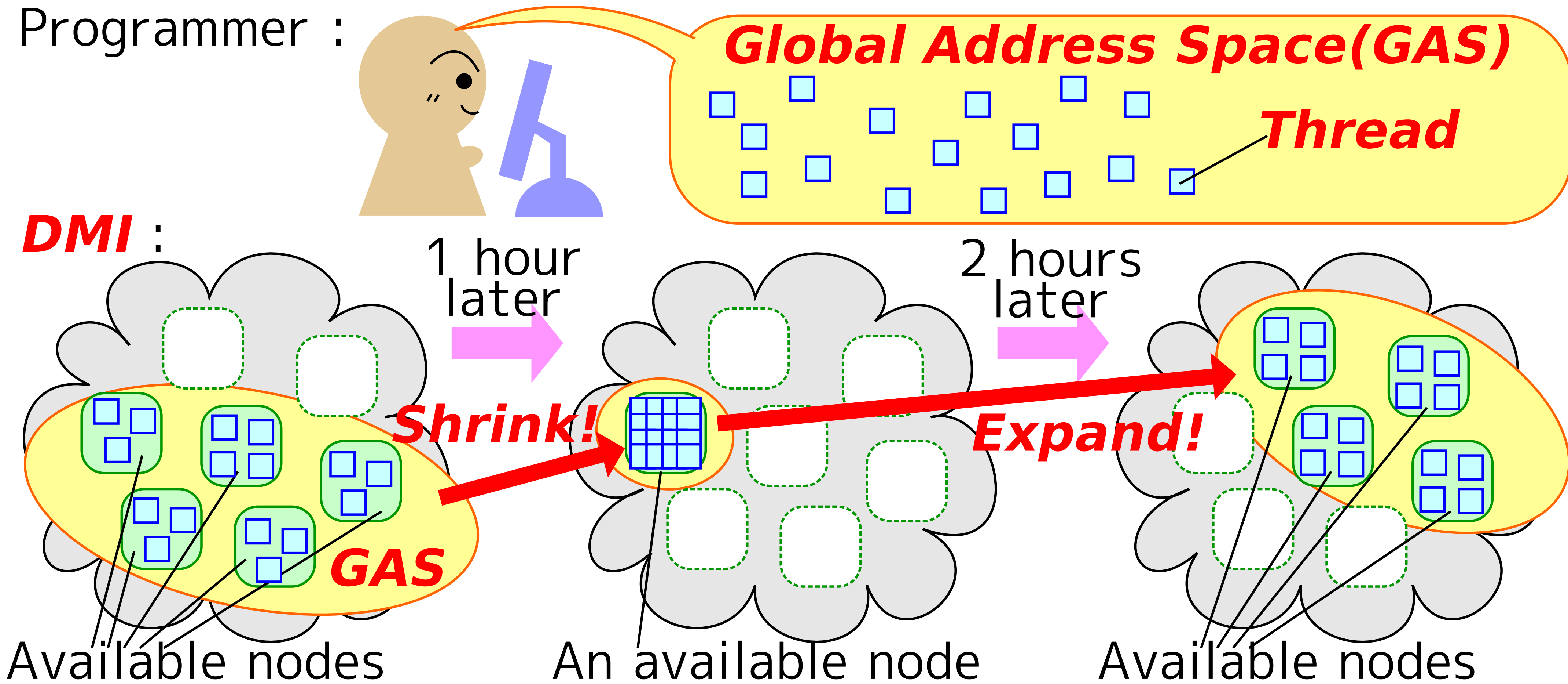


My proposal: DMI(Distributed Memory Interface)

(1) A programmer only has to create a sufficient number of threads

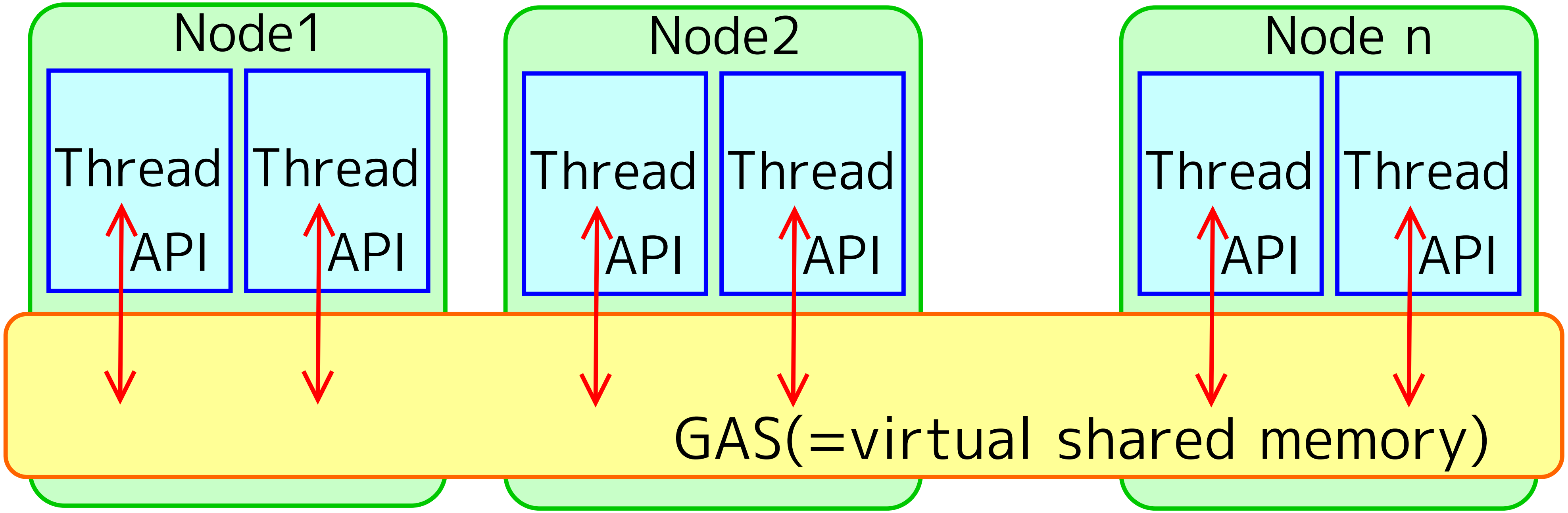
(2) A framework schedules these threads dynamically on available resources

(3) A high-performance global address space (GAS) is provided for a data sharing layer between the threads



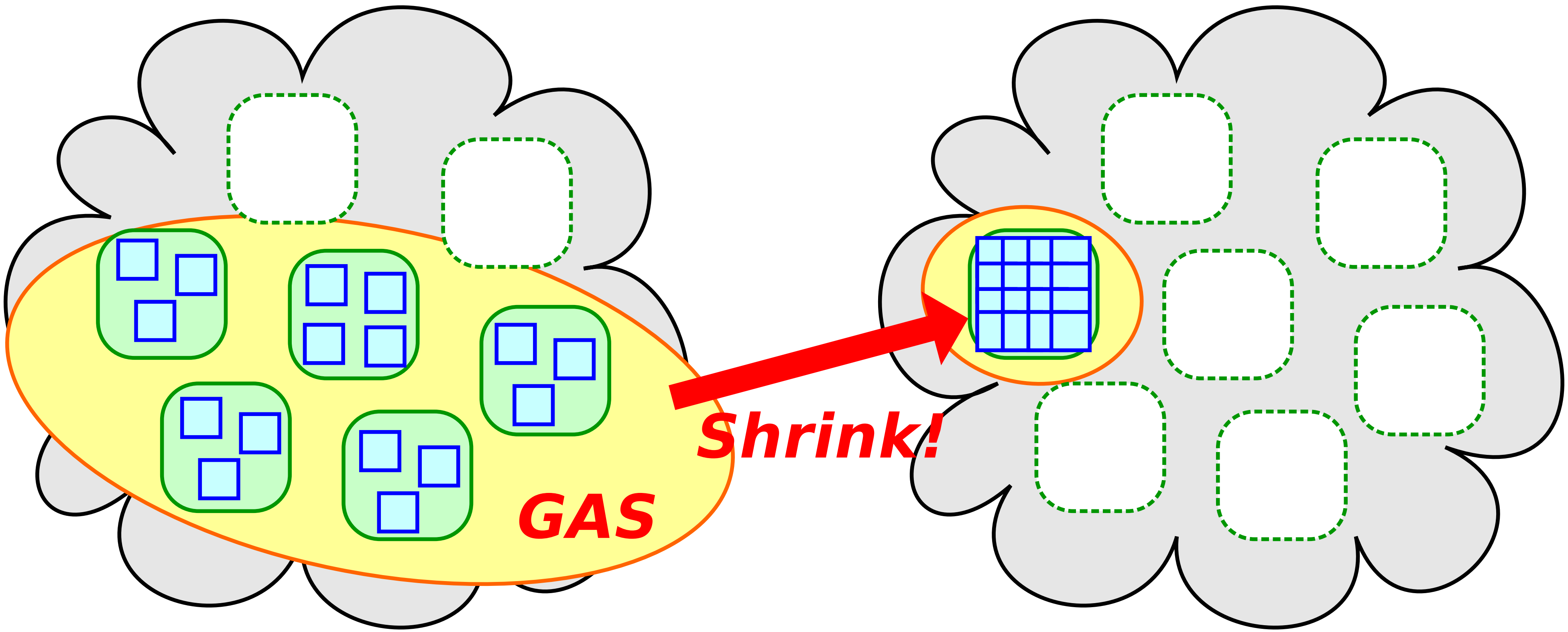
Programming interfaces of DMI

- **Similar to a (normal) shared memory environment**
  - Mmap/Munmap on the GAS
  - Read/Write from/to the GAS
  - Synchronization
  - Create/Join/Detach threads
  - ... (73 APIs in total)
- A shared library for C



Primary elemental techniques of DMI

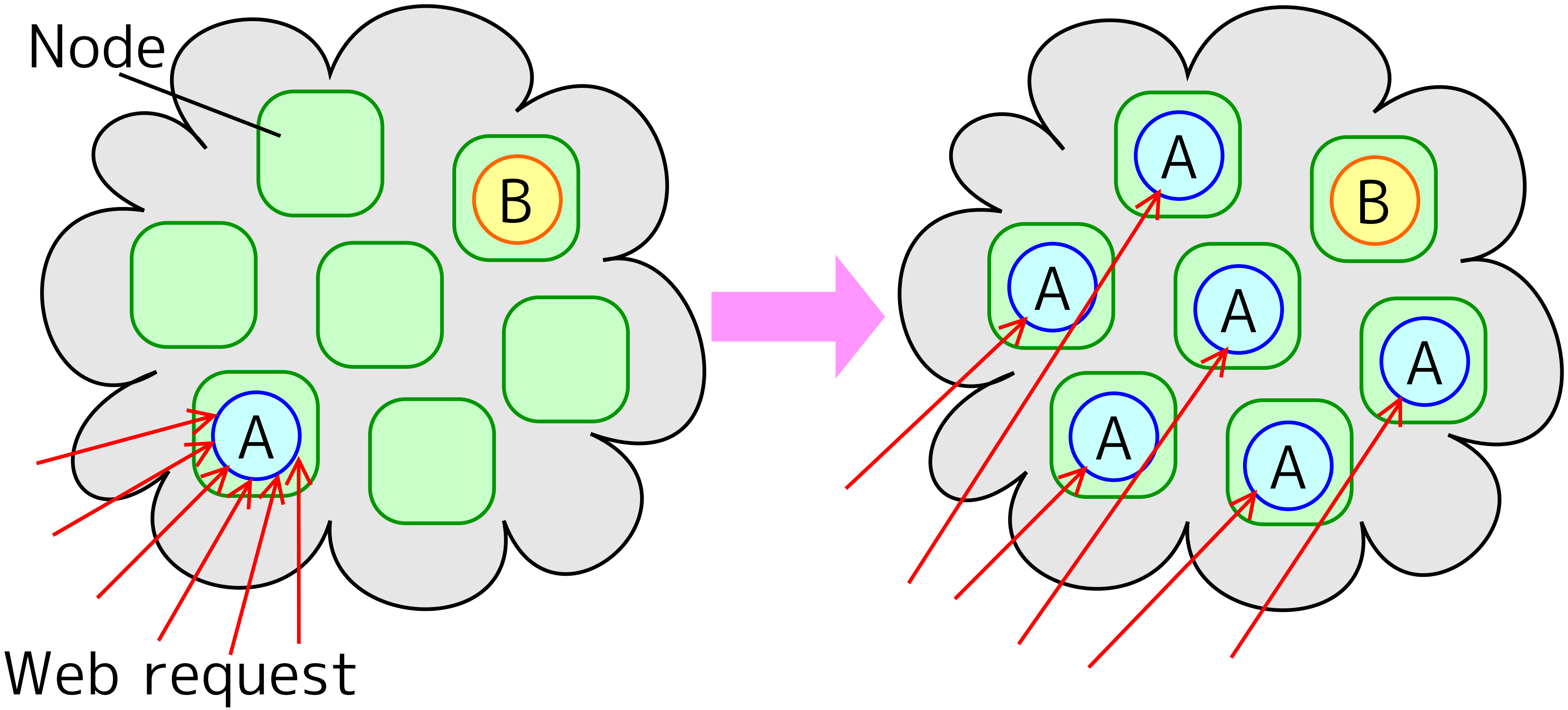
- Designing the GAS
  - How can the performance of the GAS be improved?
  - How can the GAS support dynamic joining/leaving of nodes?
- Thread migration
  - How can a live thread migrate safely?



2. Related Work

Google App Engine (GAE)

- A user can run Web apps on the Google's efficient infrastructure
- GAE **scales up/down the apps automatically and rapidly** in response to the increase/decrease of web requests
- Demerit : Each request must be processed **within 30 seconds**
  - → Almost impossible to run **long-running** large-scale parallel scientific computings



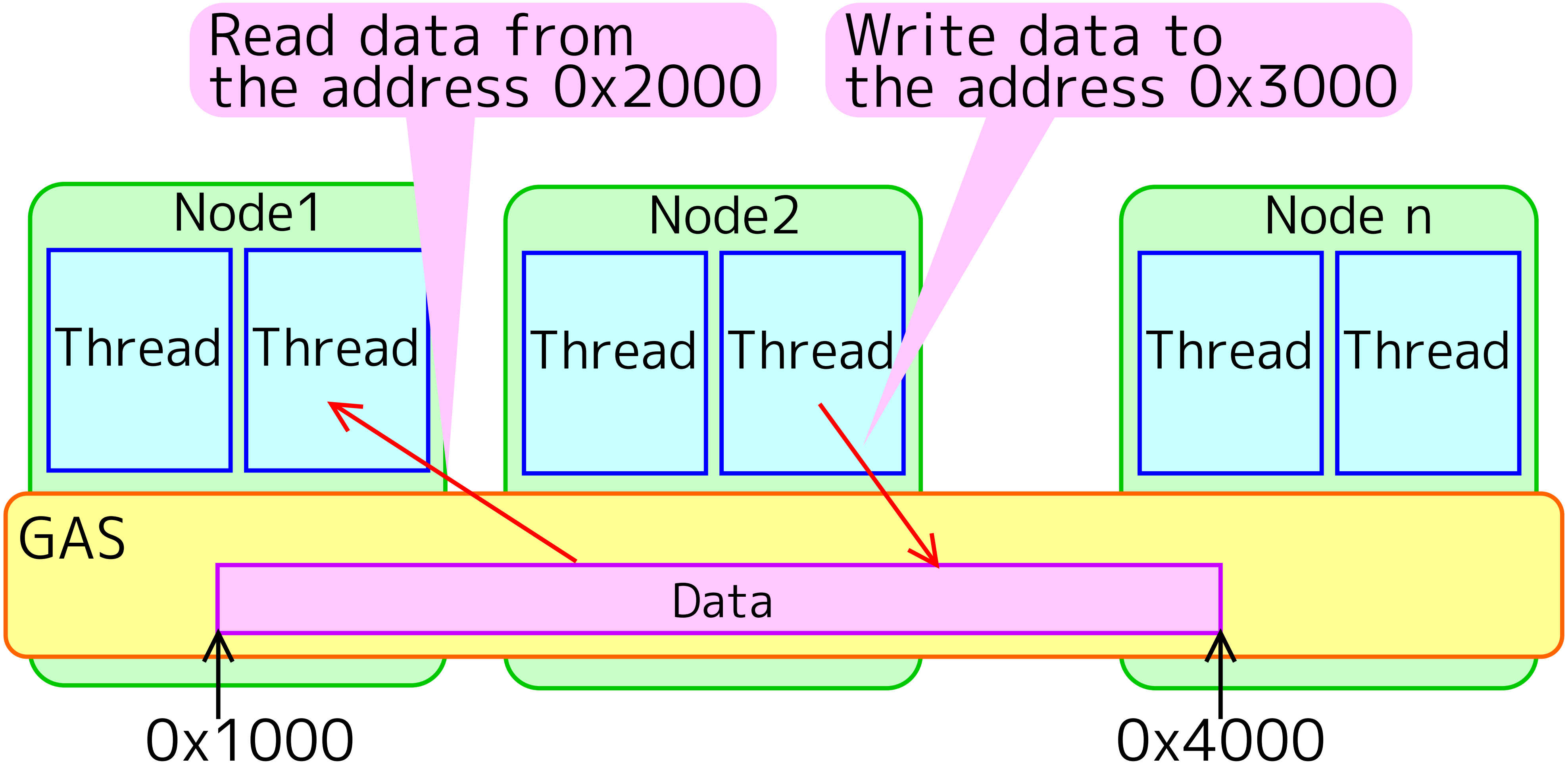
GAE vs DMI

- Requirement: Schedule resources rapidly between users
- How do GAE and DMI fulfill the requirement?
  - GAE can schedule resources only by web request
    - Hence each request must be processed in a short time
    - **Short-running** web apps
  - DMI can schedule resources (almost) anytime **by migrating threads**
    - Hence each thread can run a long time
    - **Long-running** large-scale parallel scientific computings!

3. Designing a GAS

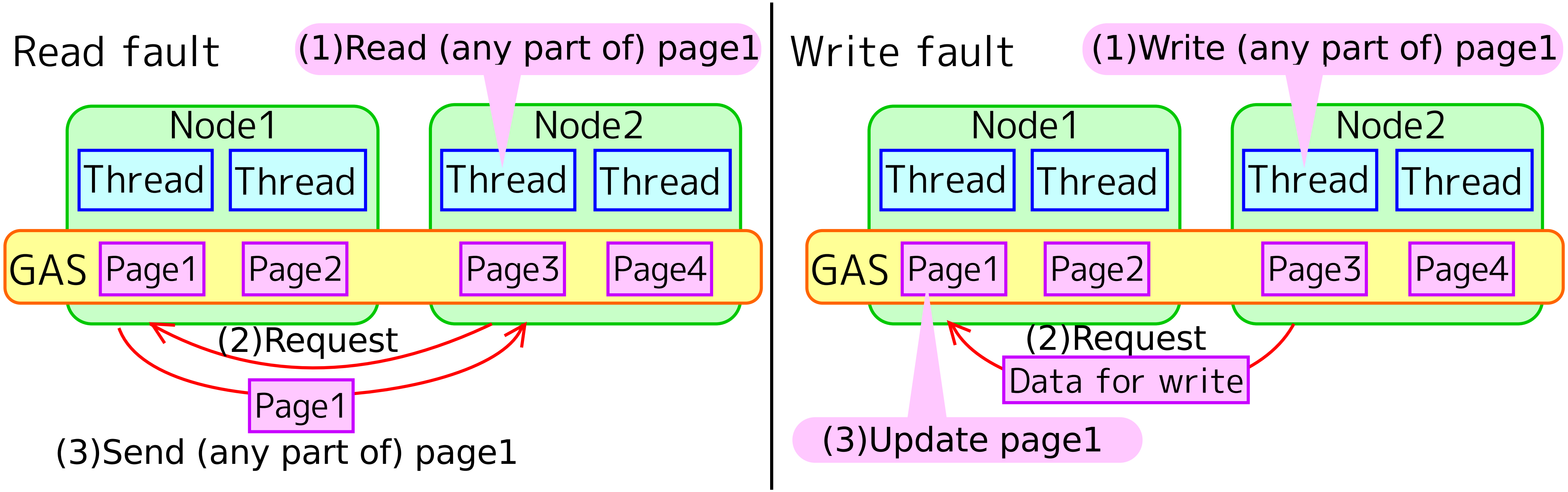
What is a GAS?

- A thread can access data on physically distributed memories by reading/writing from/to the globally unique address of the data



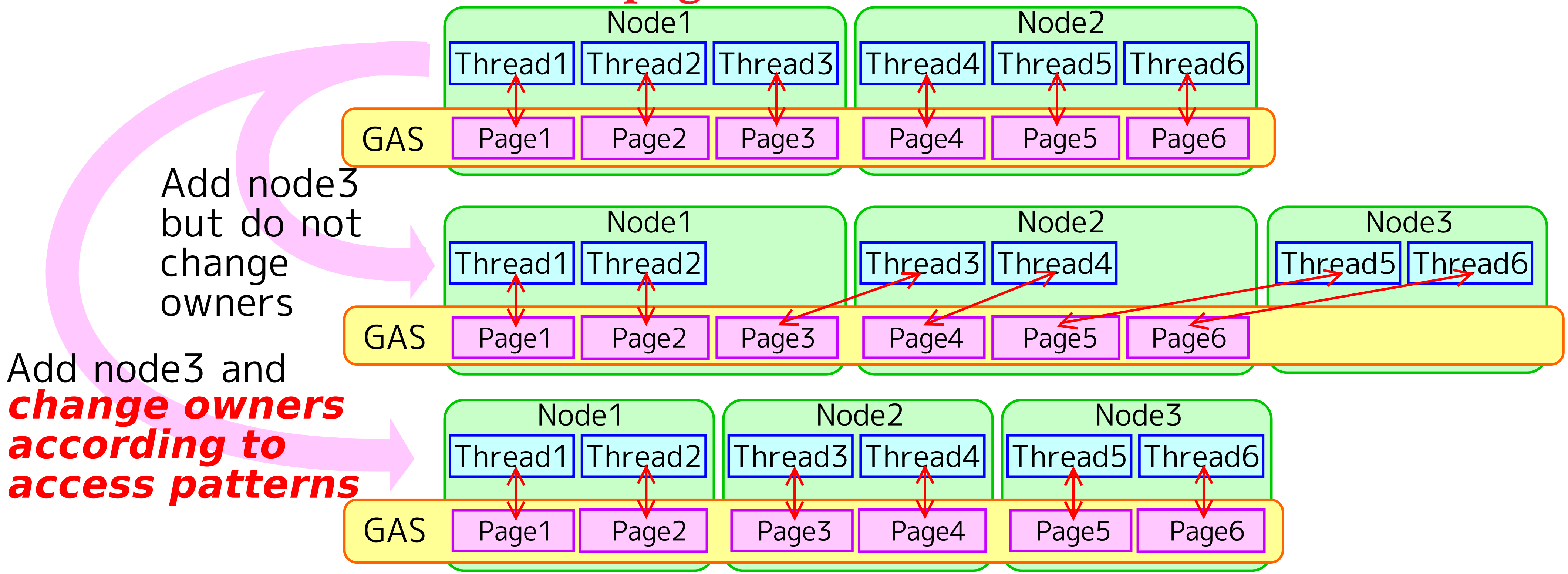
A naive implementation of the GAS

- Divide data into **pages** of suitable size (PGAS)
- Determine one fixed **owner** for each page
  - → The owner always manages the latest page and its coherency



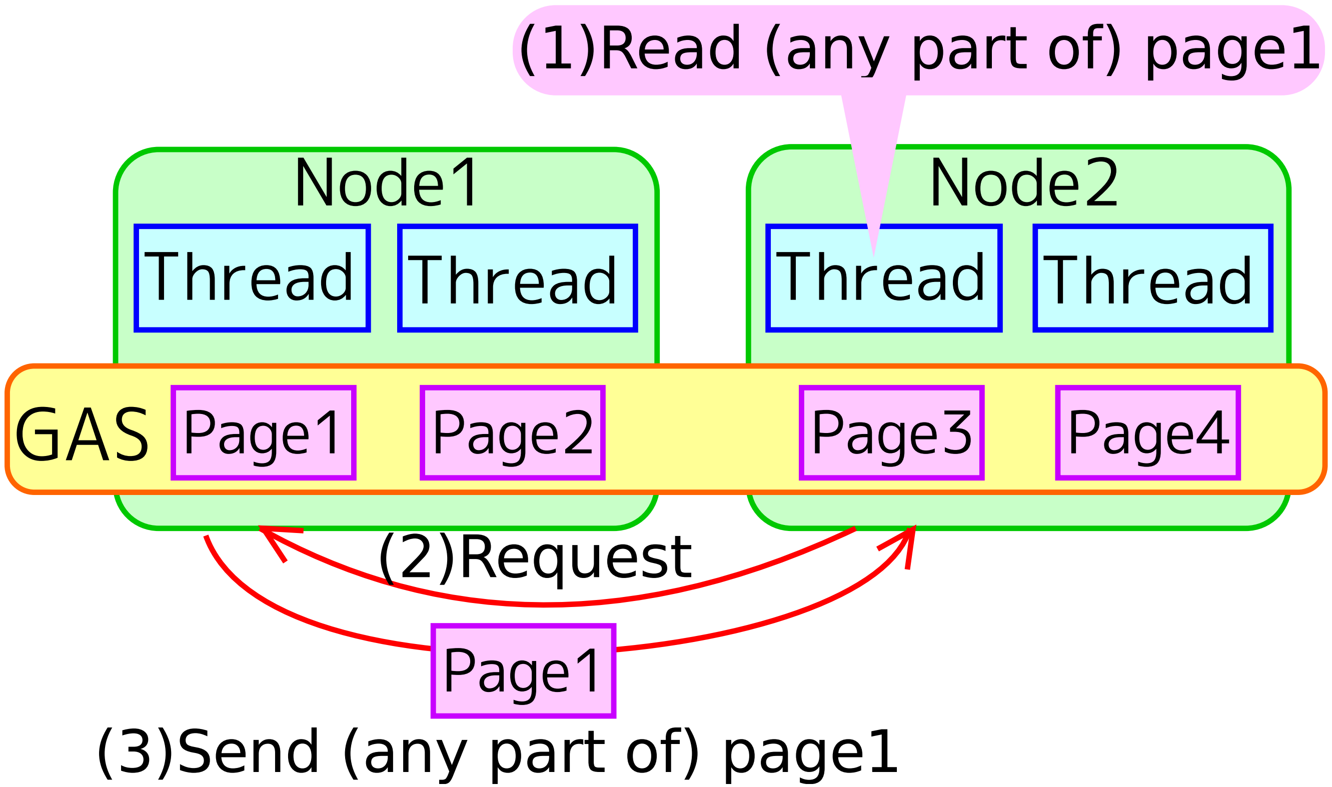
Discussion1: Should an owner be fixed?

- **If the computational scale changes, the affinity of each thread for pages also changes**
  - → An owner should migrate dynamically according to access patterns
  - → Tradeoff: But too much owner migration increases the overhead of tracing the location of the owner
- Point: Whether the owner should be fixed or not depends on **the access characteristics of the page**



Discussion2: Should a page be cached?

- It is inefficient to communicate with an owner at every read fault
  - → A page should be cached
  - → Tradeoff: But caching increases the overhead of coherency management
- Point: Whether the page should be cached or not depends on **the access characteristics of the page**



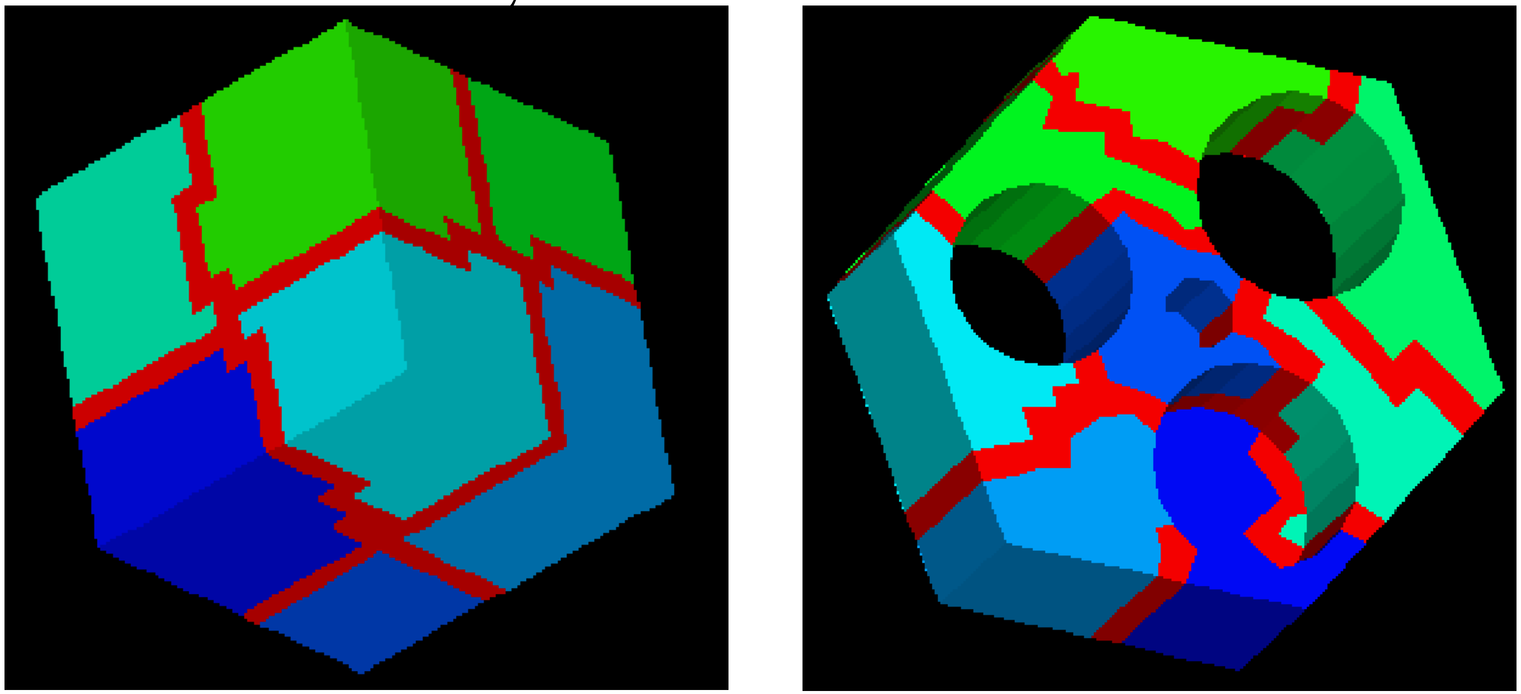
A summary and my proposal

- Summary:
  - → It is important to **allow a programmer to specify the access characteristics of each page explicitly**
- My proposal: **Selective cache read/write**
  - → A programmer can explicitly select the behavior of a page fault **at every read/write**
    - Whether an owner should be fixed or not
    - Whether a page should be cached or not
      - No cache, an invalidate cache, an update cache

```
DMI_read(gas_addr, size, buffer, SELECT);
DMI_write(gas_addr, size, buffer, SELECT);
```

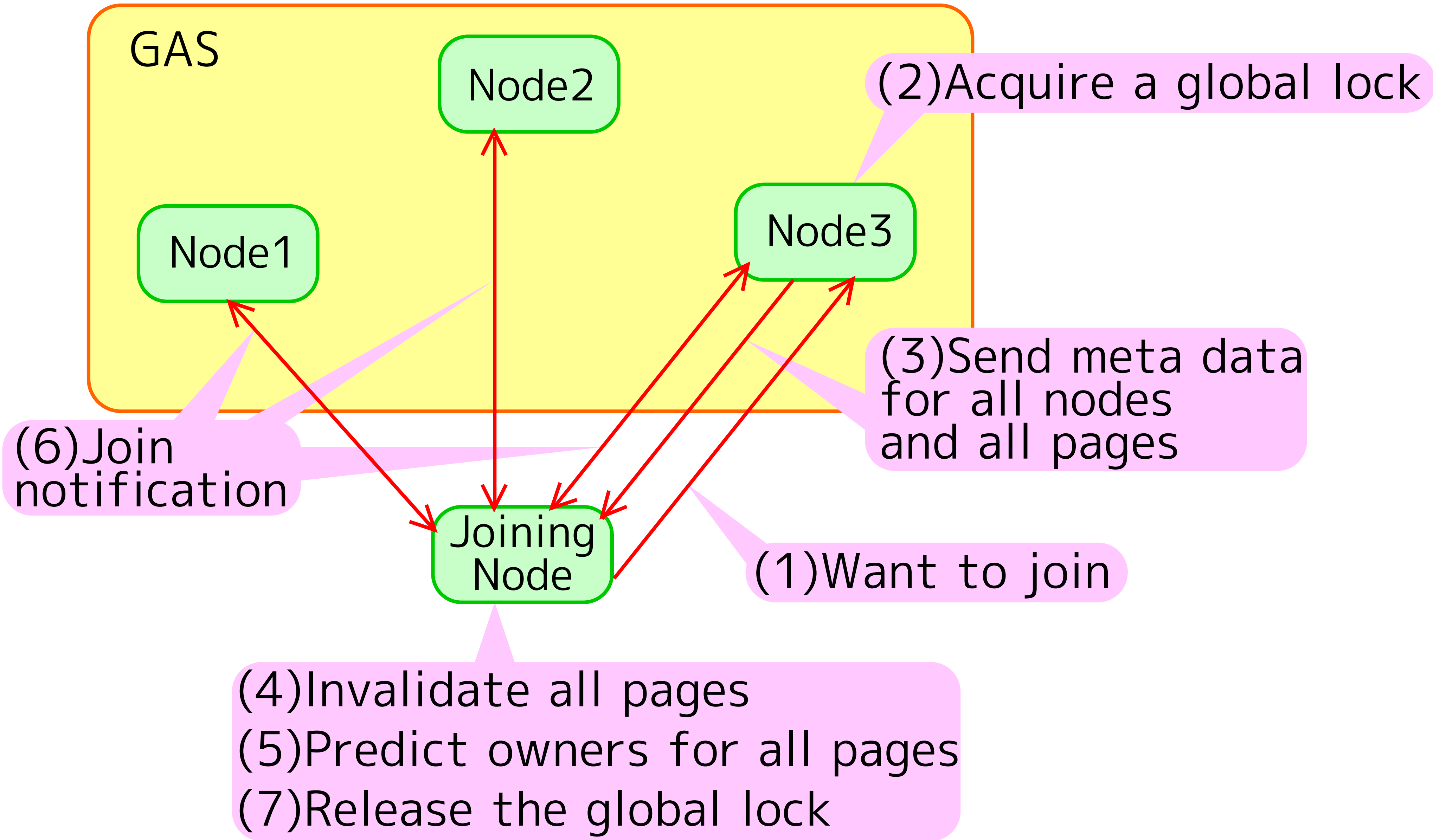
Other optimization methods

- Productive APIs for communicating the values of boundary points in parallel scientific computings with domain decompositions
- Automatic load balancing of data transfers
- Aggregation of discrete accesses
- Asynchronous read/write
- User-defined page size
- User-defined read-modify-write

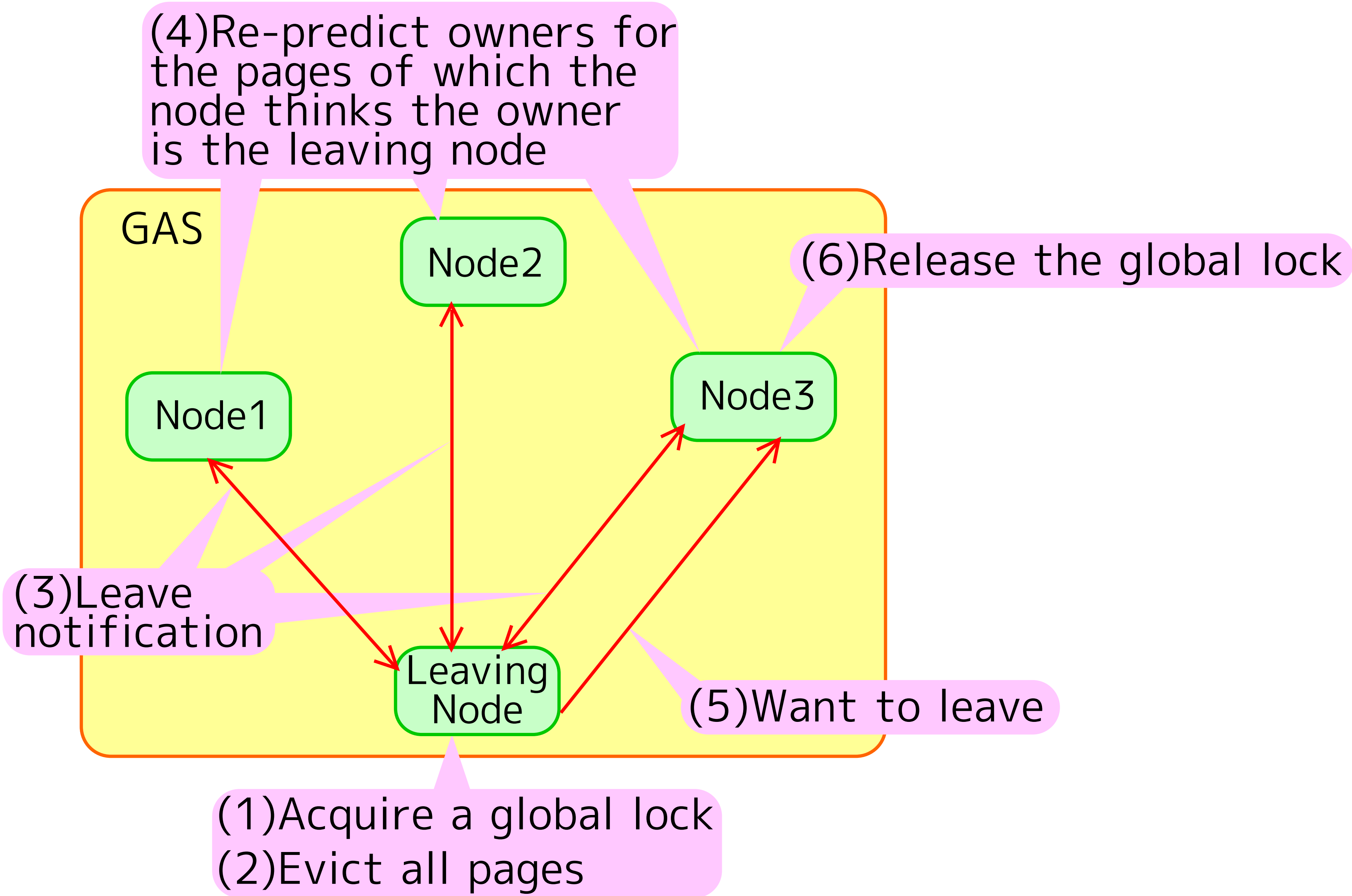


[Nakajima,2009]

Joining of a node to the GAS



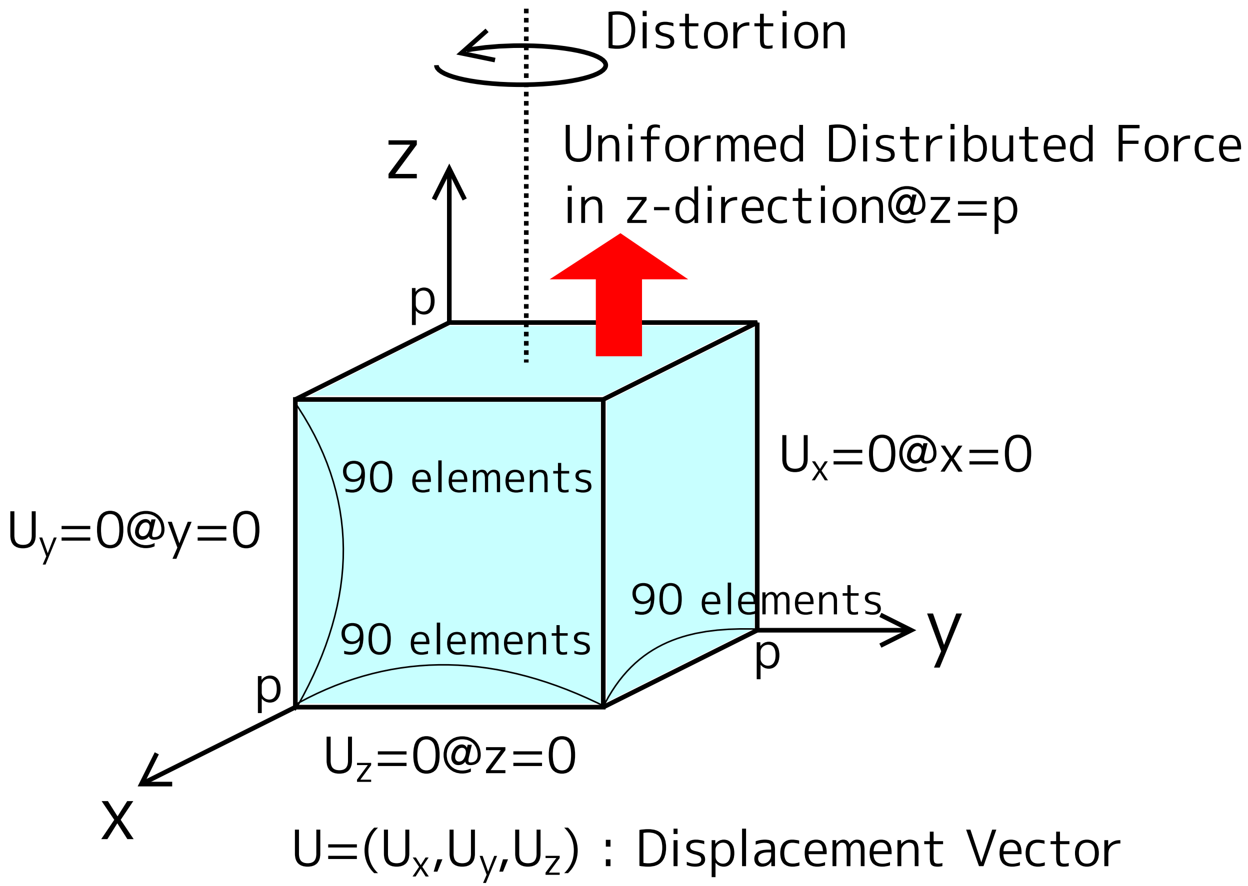
Leaving of a node from the GAS

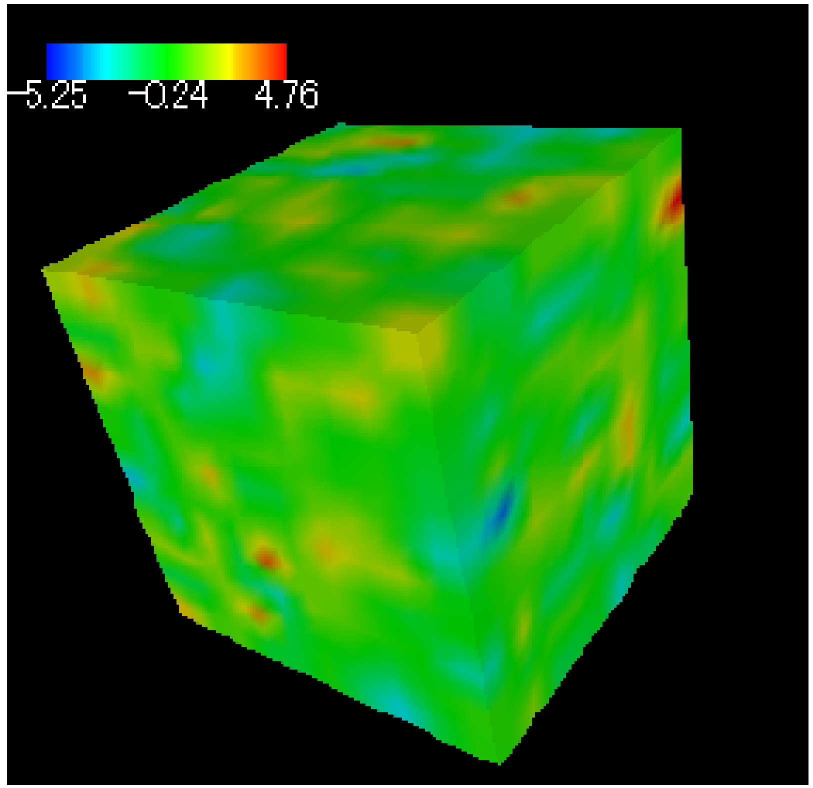


- DMI defines protocols for consistency maintenance strictly

An experiment: An FEM

- A stress analysis using an FEM
  - → A hard-to-converge problem based on the **real-world** engineering
  - → The problem used in the programming contest on supercomputers
- Environment: 8 cores × 16 nodes, 1GbitE

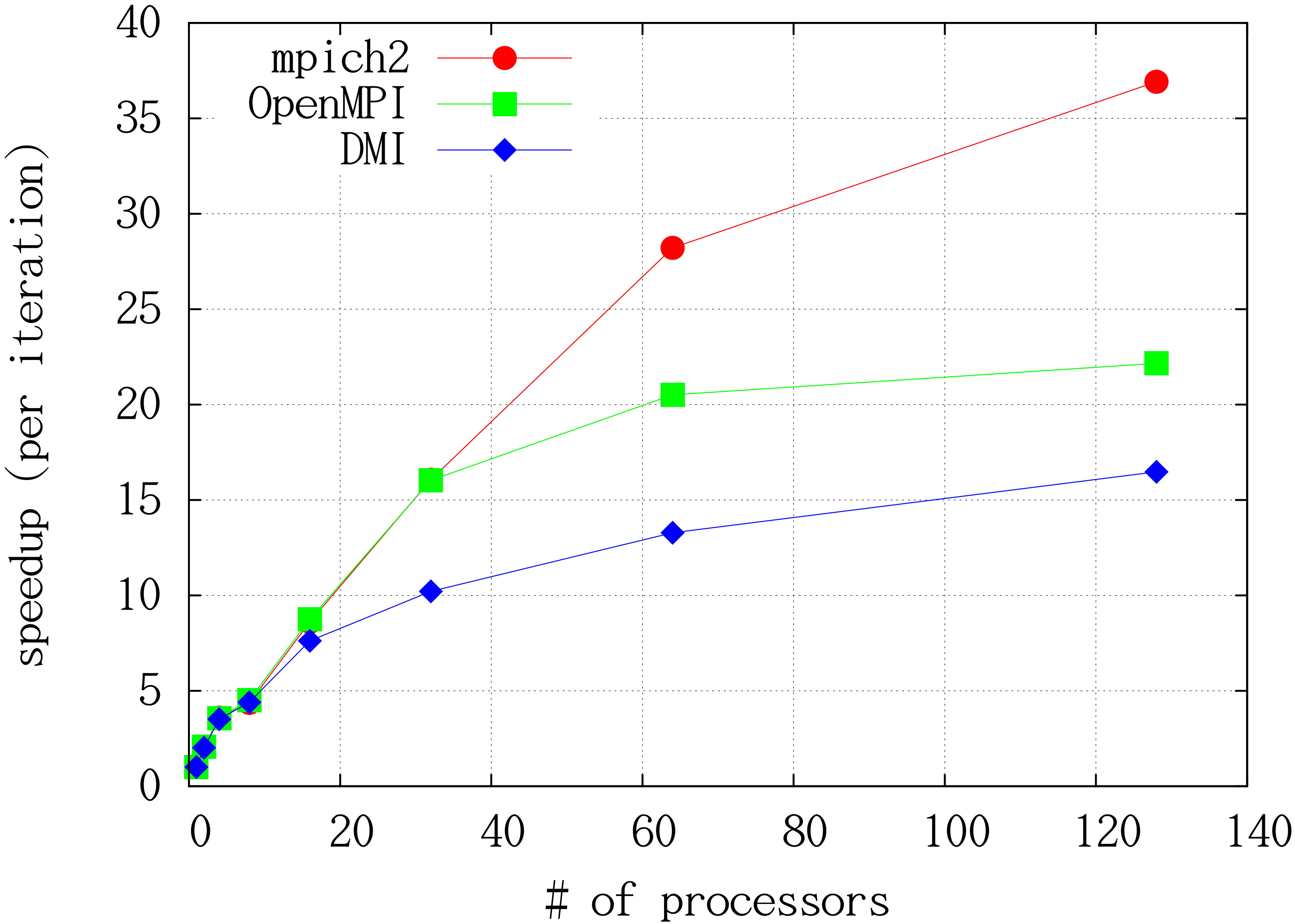




[Nakajima,2009]

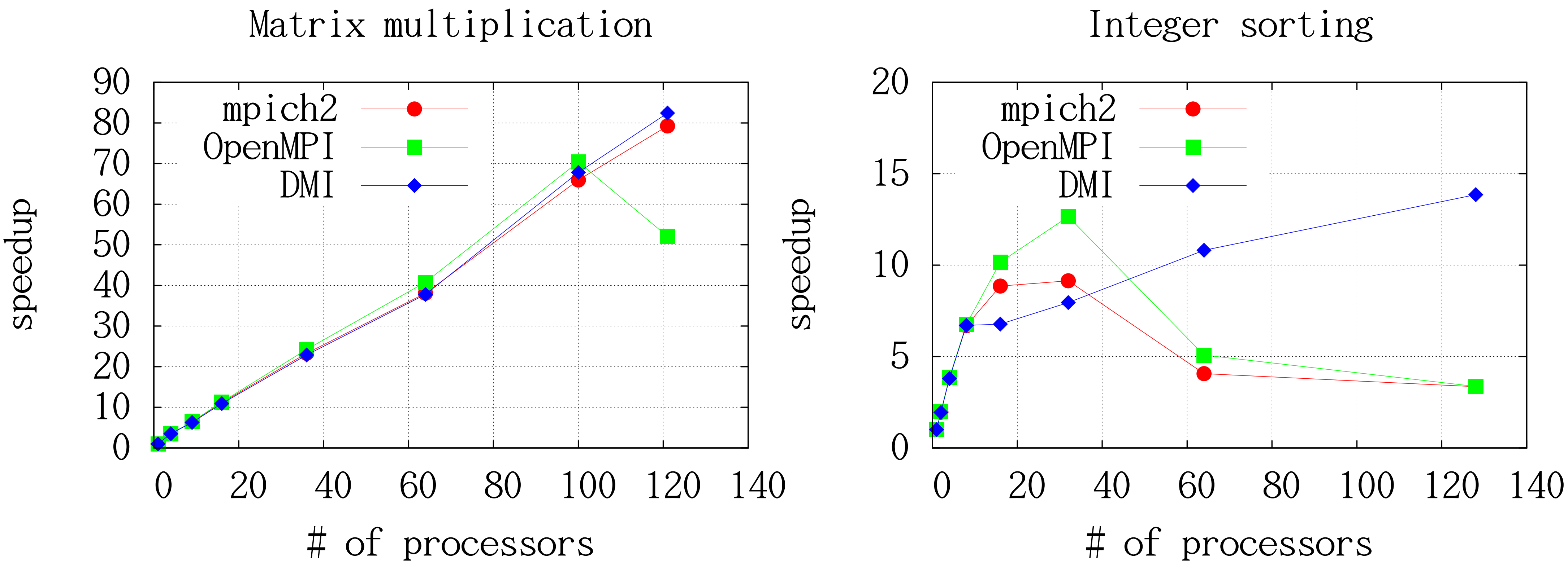
The result: The scalability of the FEM

- Speedup = (the execution time when executed using 1 processor) / (the execution time when executed using n processors)



The result: The scalability of other apps

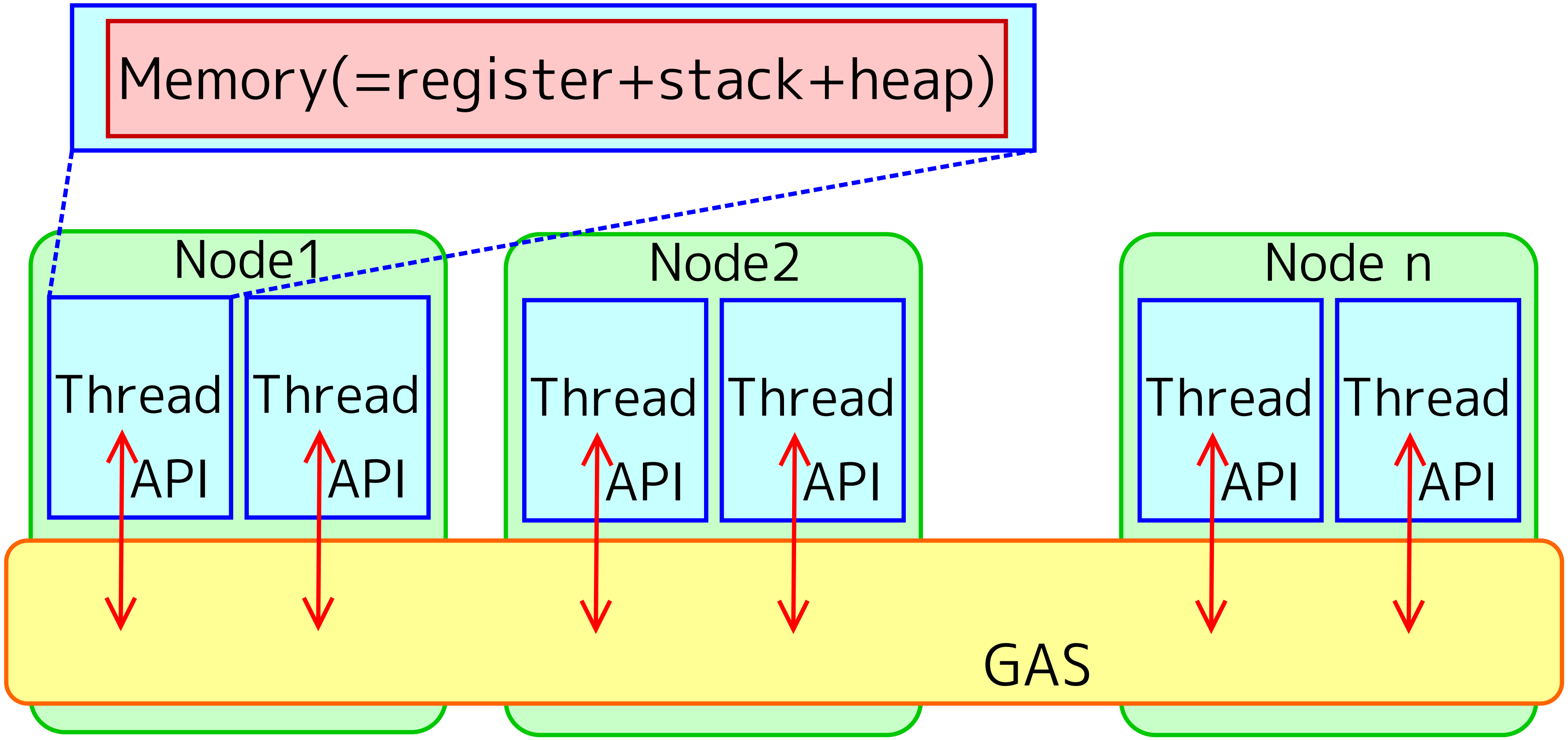
- Matrix multiplication
- Integer sorting



❖ 4. Thread Migration

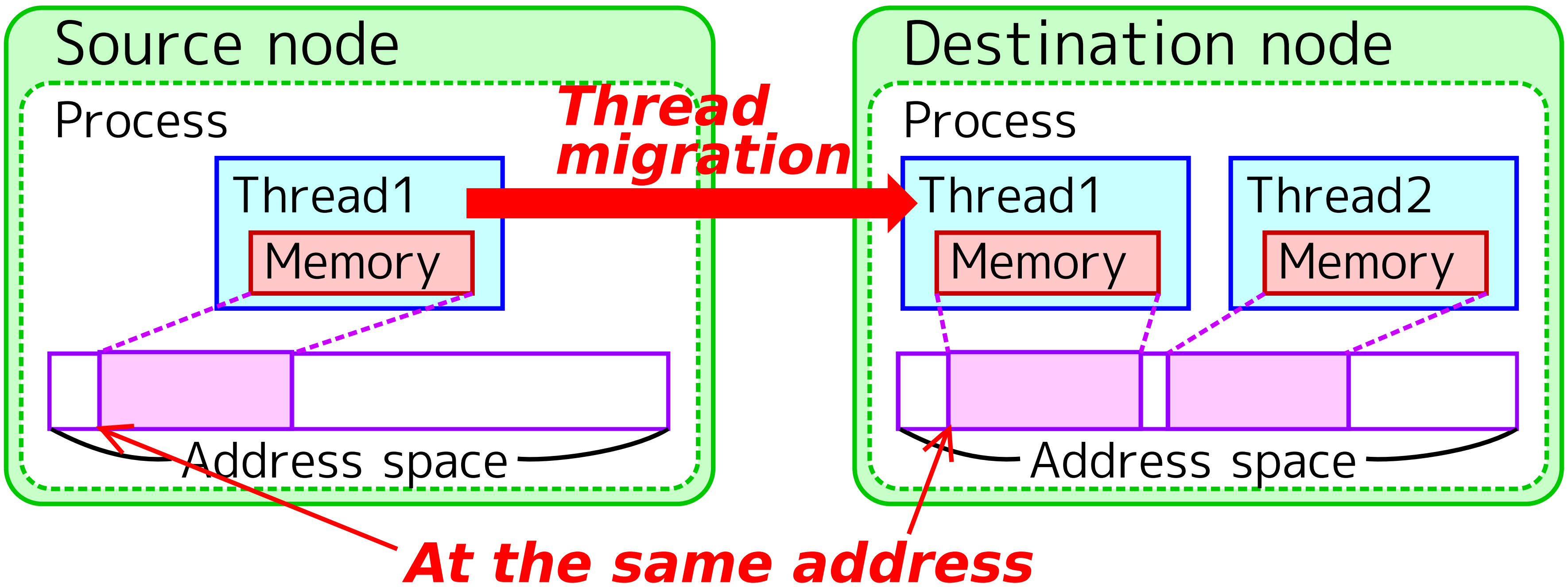
Assumptions: Threads of DMI

- Each process has multiple threads
- Memory of a thread = Register + Stack + Heap
- Each thread just accesses the memory of the thread
  - → Data sharing between threads is achieved through a GAS
  - → No file I/O, no network I/O



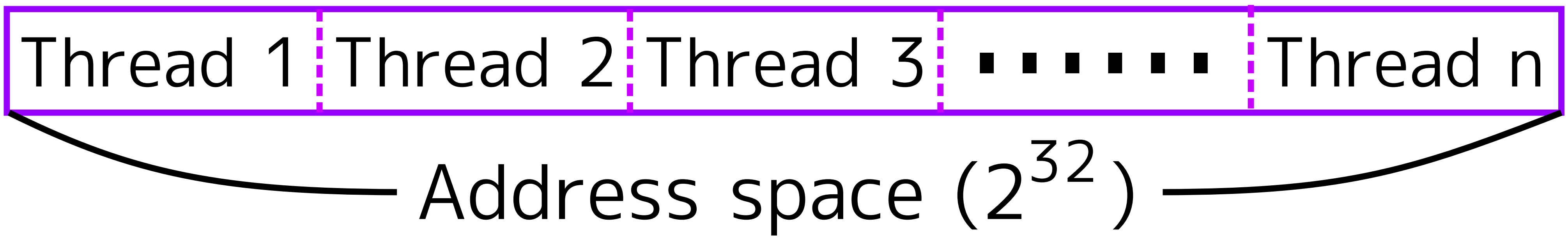
Thread migration in DMI

- Thread migration:
  - Stop a thread on the source node
  - Migrate the memory of the thread
  - Resume the thread on the destination node
- To avoid pointer invalidation, **the memory of the thread must be allocated on the same address**[Antoniu et al,1999]
  - But there is no guarantee that the appropriate addresses are not used at the destination node



How can memory be allocated on the same address?

- An existing approach[Weissman et al,1998]:
  - Divide the whole address space (ex.2^{32}) and fix statically the addresses that each thread can use
  - Guarantee the global uniqueness of the addresses used by each thread
- "This is impractical in a 32bit arch, but is **practical in a 64bit arch**"[Itzkovitz et al,1998][Weissman et al,1998][Thitikamol et al,1999]



Is it really practical in a 64bit arch?

- The size of the address space of most 64bit arches is 2^{47}

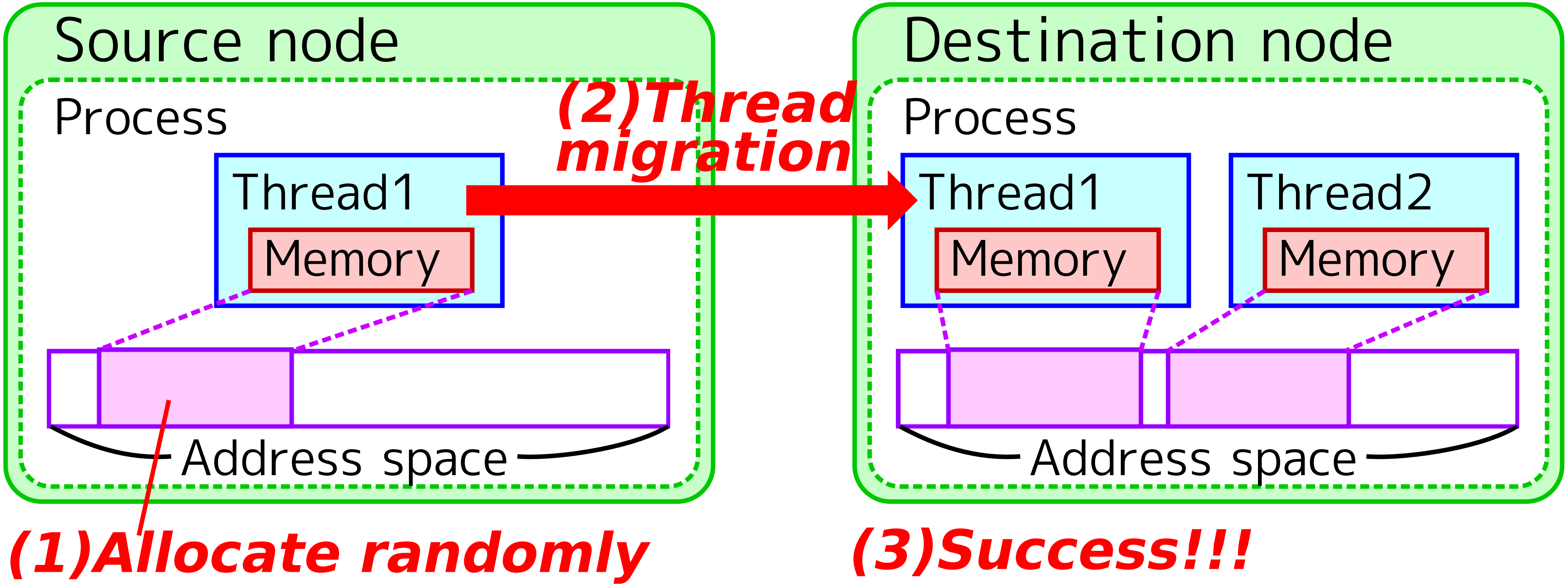
	The number of threads		The memory size that each thread can use
2^{47} bytes =	8192	×	64GB
2^{47} bytes =	1024	×	512GB

- "The limit is approaching!"
  - → **Thread migration unrestricted by the size of the address space** is required

My proposal: Random-address(1)

(1) Determine the addresses used by each thread **randomly**

(2) If we are lucky, addresses do not collide when a thread migrates

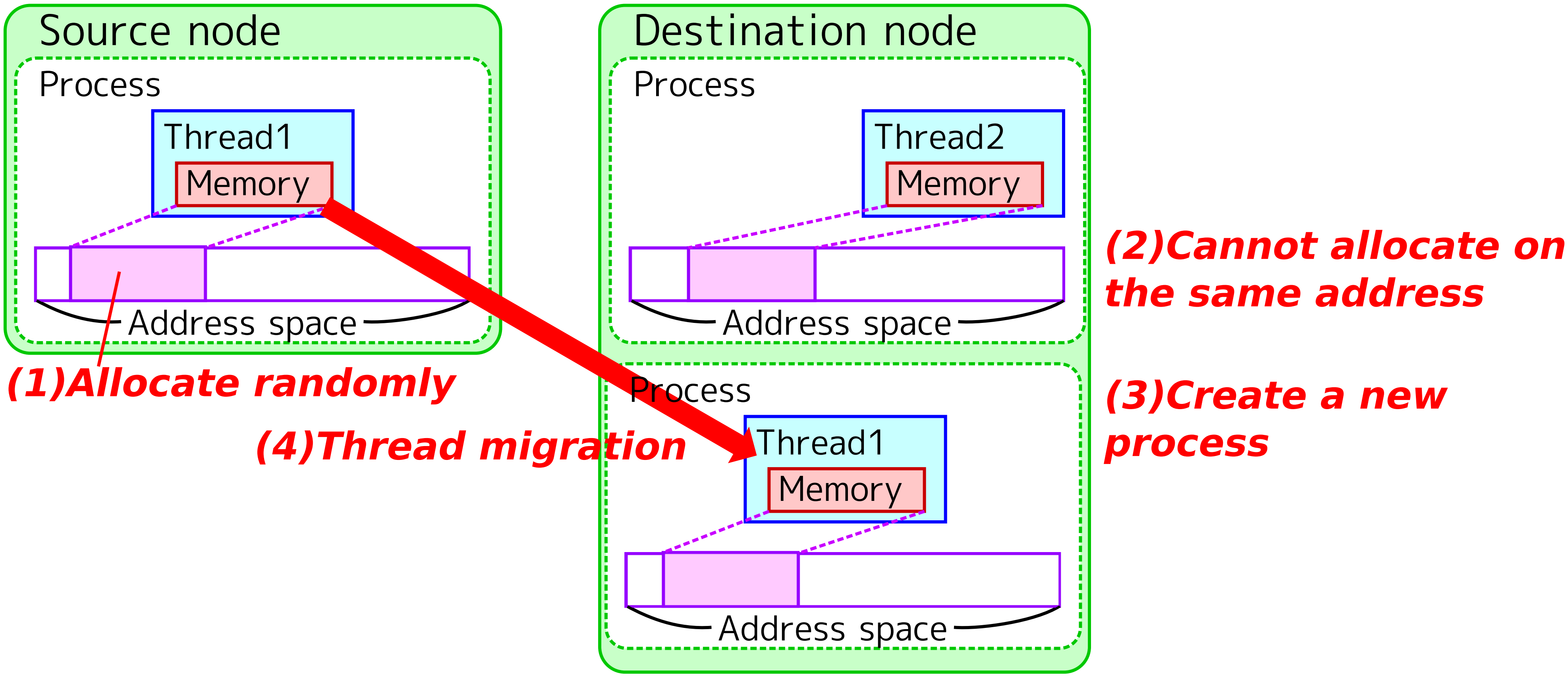


My proposal: **Random-address**(2)

(3) If we are not lucky, addresses collide

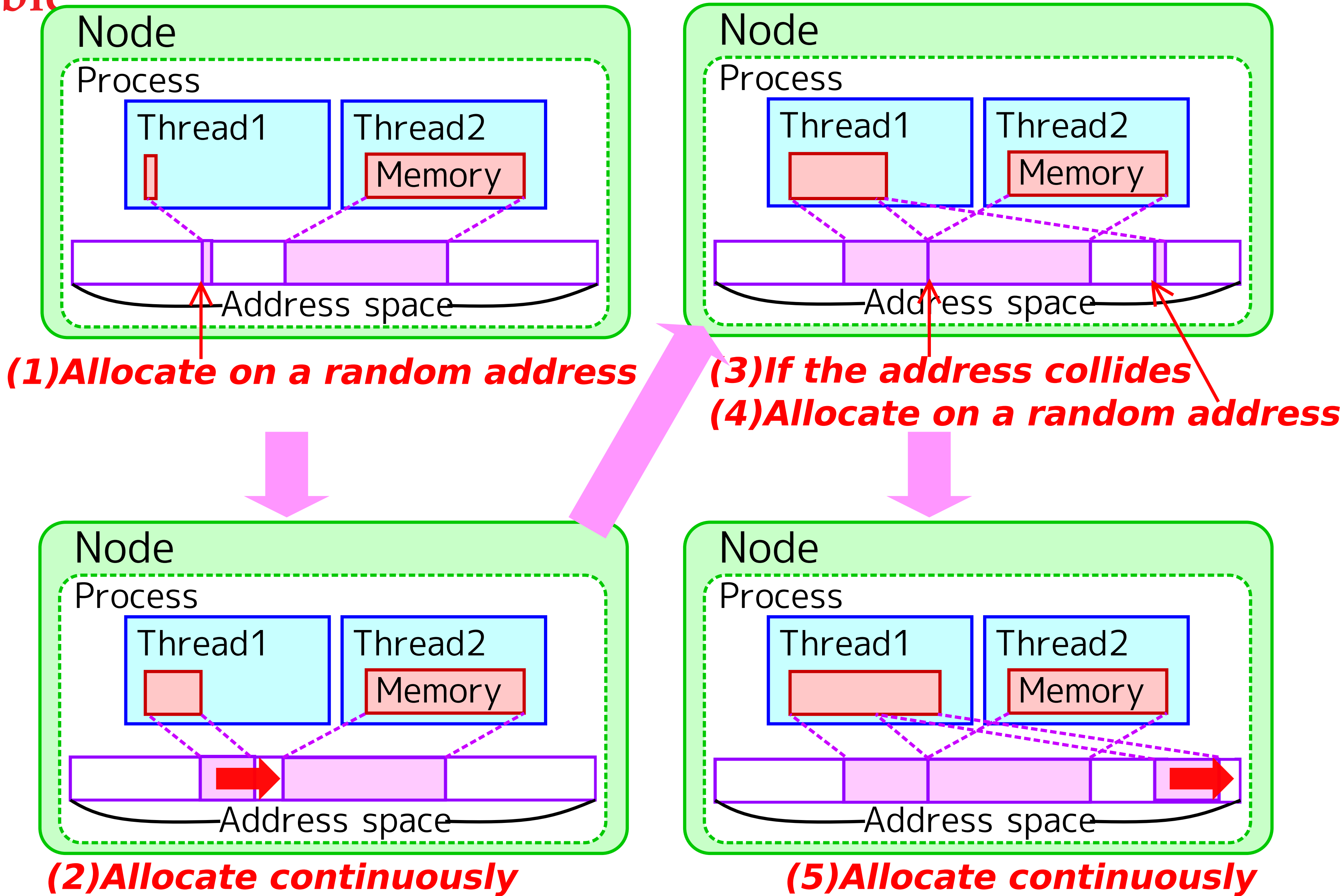
- **(a)** Then, **create a new process** (=a new address space) on the destination node
- **(b)** **Migrate the thread into the new process**

➤ Note: This approach cannot be achieved without the GAS supporting dynamic joining of nodes



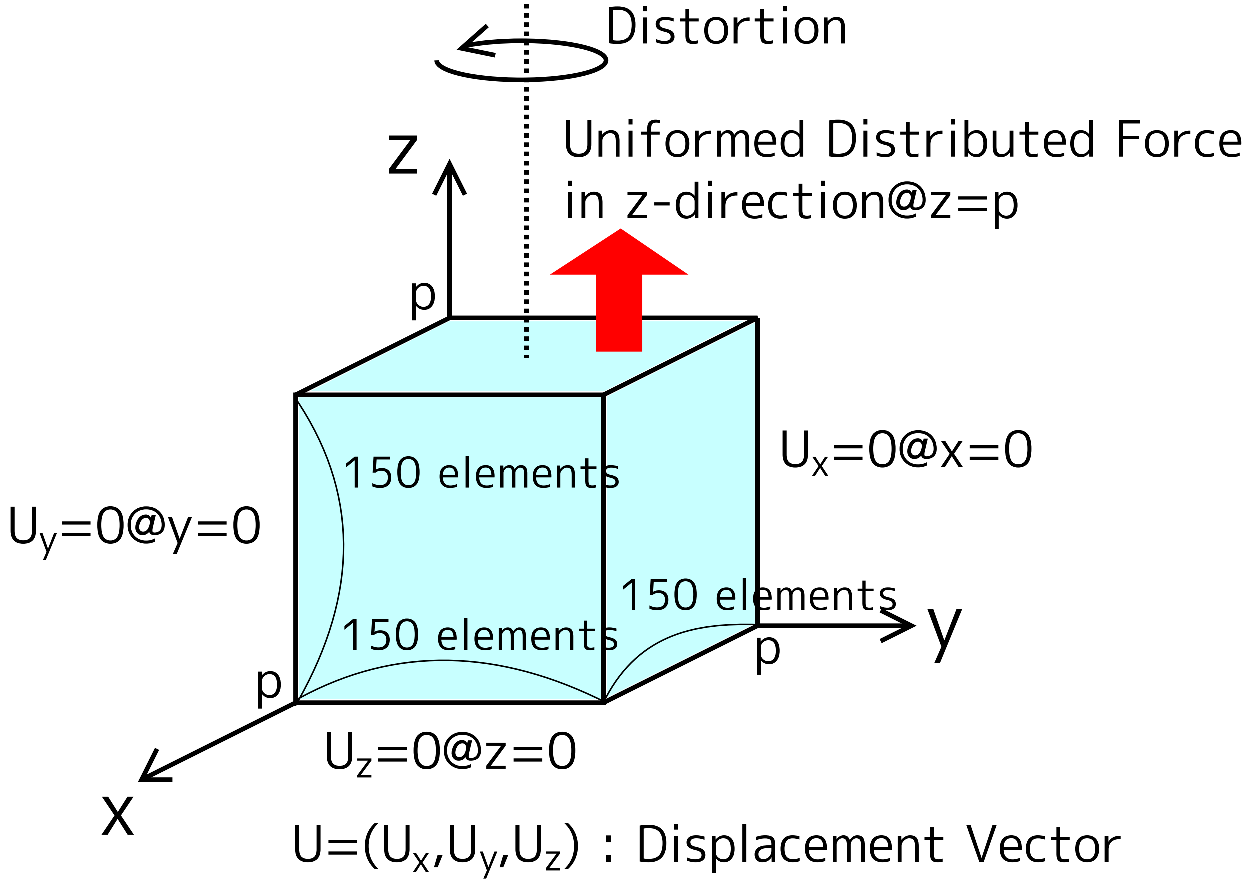
How to minimize the probability of the address collision

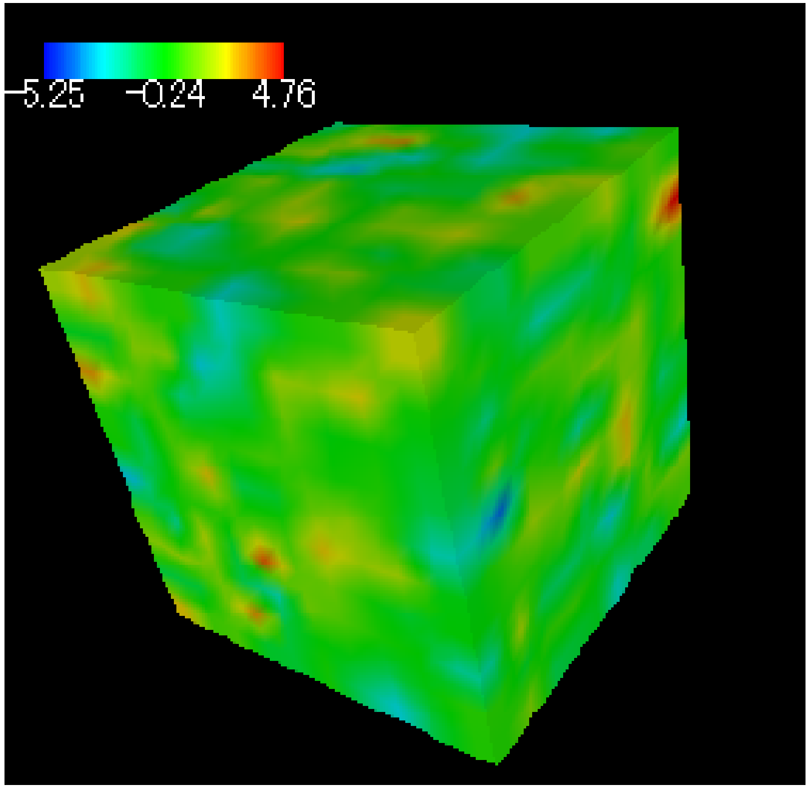
- One of the optimized solutions: "**Use addresses as continuous as possible**"



An experiment: An FEM(1)

- A stress analysis using an FEM
- Environment: 8 cores × 16 nodes, 10GbitE





[Nakajima,2009]

An experiment: An FEM(2)

- Repeat until convergence
- **Only** insert a chance of cooperative thread migration at the head of each iteration

solve $A\vec{x} = \vec{b}$:

K = preconditioned matrix of A

$\vec{r_0} = \vec{b} - A\vec{x}$

initialize vectors $\vec{x_0}, \vec{r_0}, \vec{r_0^*}, \vec{p_0}, \vec{u_0}, \vec{y_0}, \vec{v_0}$ properly

initialize $\beta_{-1}, \xi_0, \eta_0$ properly

for $n = 0, 1, 2, \ldots$ **until convergence do**

DMI_yield()

$\vec{p_n} = K^{-1}\vec{r_n} + \beta_{n-1}(\overrightarrow{p_{n-1}} - \overrightarrow{u_{n-1}})$

$A\vec{p_n} = AK^{-1}\vec{r_n} + \beta_{n-1}(A\overrightarrow{p_{n-1}} - A\overrightarrow{u_{n-1}})$

$\alpha_n = (\vec{r_0^*}, \vec{r_n})/(\vec{r_0^*}, A\vec{p_n})$

$\xi_n = ((\vec{y_n}, \vec{y_n})(\vec{v_n}, \vec{r_n}) - (\vec{y_n}, \vec{r_n})(\vec{v_n}, \vec{y_n}))/((\vec{v_n}, \vec{v_n})(\vec{y_n}, \vec{y_n}) - (\vec{y_n}, \vec{v_n})(\vec{v_n}, \vec{y_n}))$

$\eta_n = ((\vec{v_n}, \vec{v_n})(\vec{y_n}, \vec{r_n}) - (\vec{y_n}, \vec{v_n})(\vec{v_n}, \vec{r_n}))/((\vec{v_n}, \vec{v_n})(\vec{y_n}, \vec{y_n}) - (\vec{y_n}, \vec{v_n})(\vec{v_n}, \vec{y_n}))$

$\vec{u_n} = K^{-1}(\xi_n A\vec{p_n} + \eta_n \vec{y_n}) + \eta_n \beta_{n-1}\overrightarrow{u_{n-1}}$

$\vec{z_n} = \xi_n K^{-1}\vec{r_n} + \eta_n \overrightarrow{z_{n-1}} - \alpha_n \vec{u_n}$

$\overrightarrow{y_{n+1}} = \xi_n AK^{-1}\vec{r_n} + \eta_n \vec{y_n} - \alpha_n A\vec{u_n}$

$\overrightarrow{x_{n+1}} = \vec{x_n} + \alpha_n \vec{p_n} + \vec{z_n}$

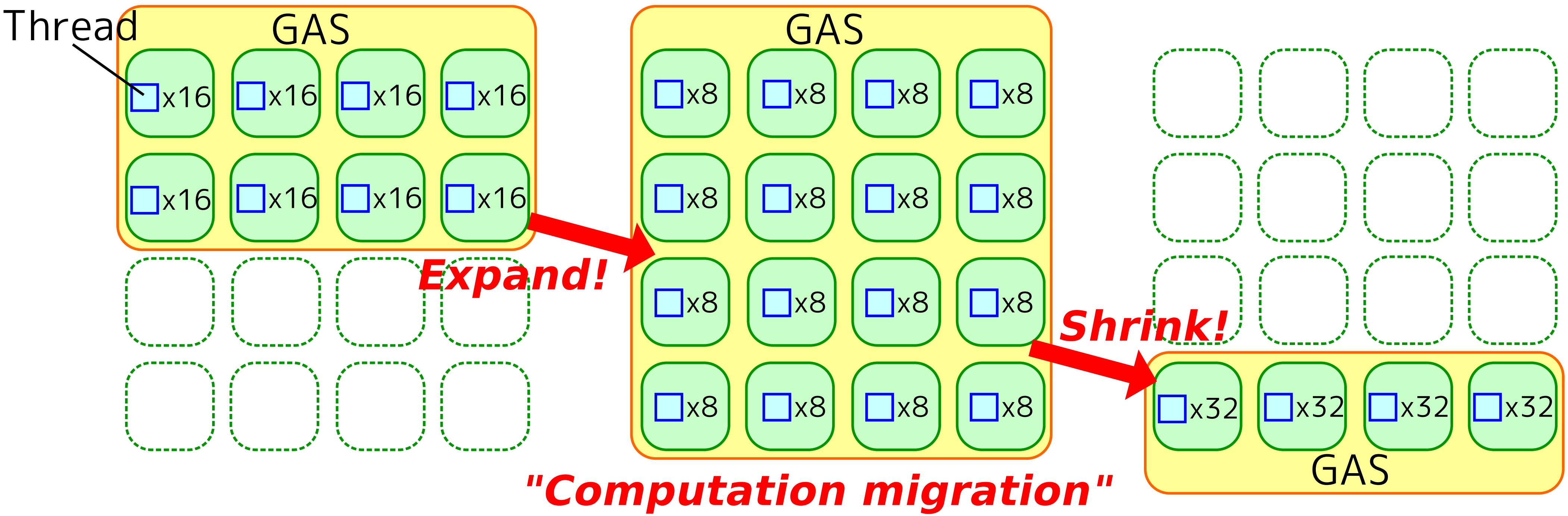
$\overrightarrow{r_{n+1}} = \vec{r_n} - \alpha_n A\vec{p_n} - \overrightarrow{y_{n+1}}$

$\beta_n = (\alpha_n/\xi_n)(\vec{r_o^*}, \overrightarrow{r_{n+1}}/(\vec{r_0^*}, \vec{r_n}))$

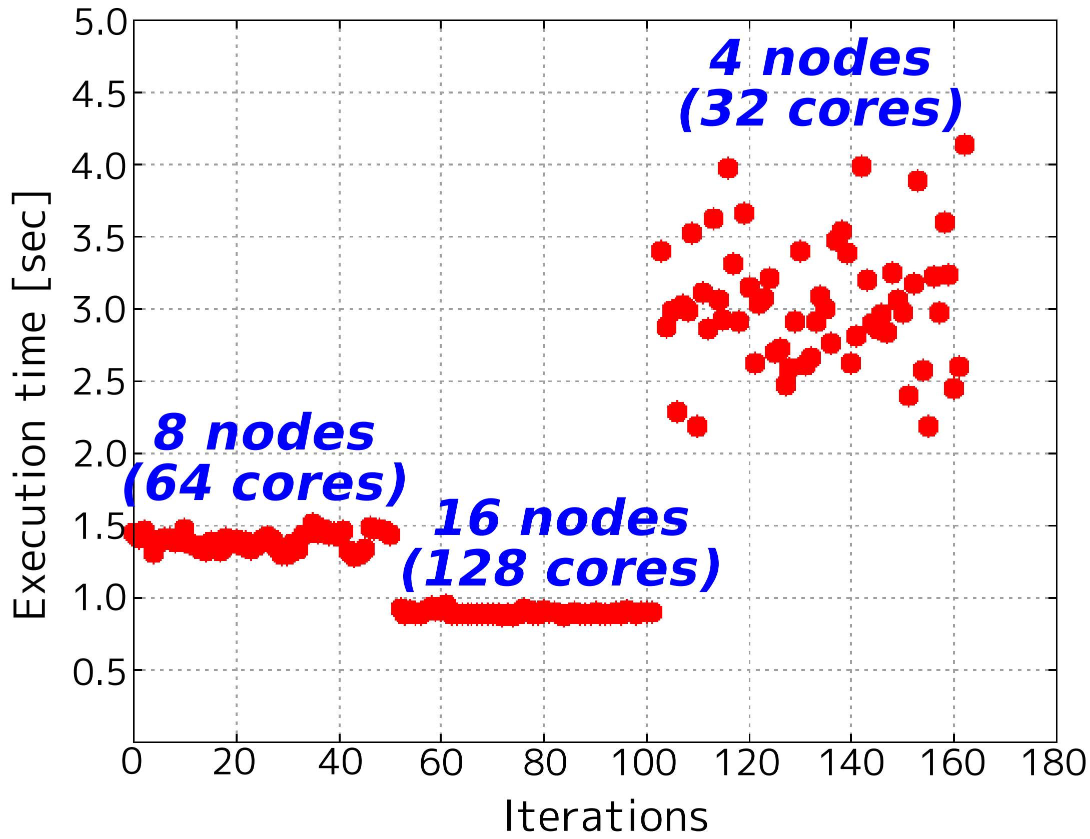
endfor

An experiment: An experimental scenario

- Create 128 threads
  - Each thread consumes 500MB memory (64GB in total)
  - A GAS consumes 335MB memory
- Change available resources:
  - (1) Run on the nodes 1∼8
  - (2) Add the nodes 9∼16
  - (3) Remove the nodes 1∼12



The result: Dynamic change of parallelism

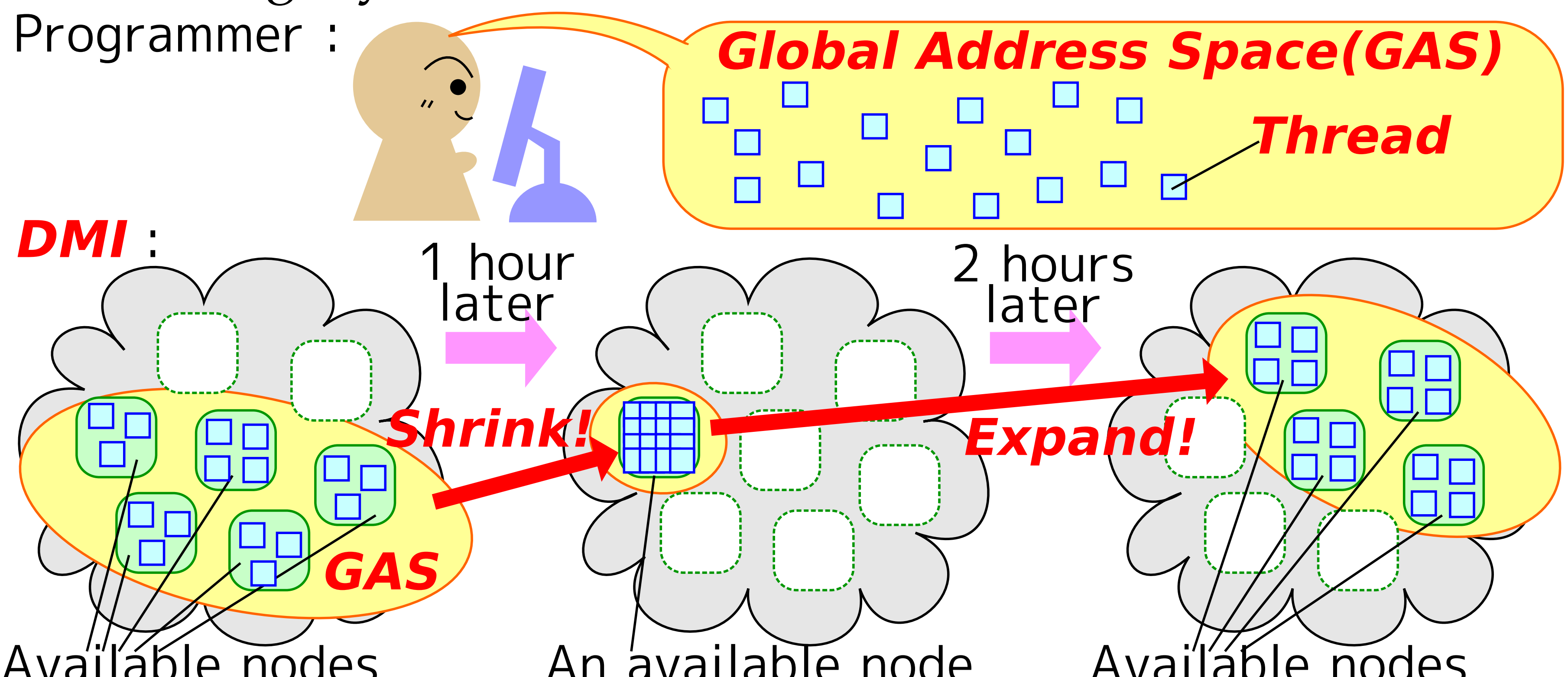


- **DMI expanded and shrinked the computational scale dynamically in response to the change of available resources**
- No address collision happened
- Migration time:
  - 17 sec for adding 8 nodes, migrating 120 threads (57GB memory)
  - 30 sec for removing 12 nodes, migrating 120 threads (57GB memory)

❖ 5. Conclusions

A summary

- **DMI(Distributed Memory Interface)**: A PGAS framework for a parallel computation on a cloud
  - → **A programmer only has to create a sufficient number of threads**
  - → **A framework schedules these threads dynamically on available resources**
  - → **A high-performance global address space (GAS)** is provided for a data sharing layer between the threads



Future work

- Evaluate and optimize **real-world** scientific computings on the **real-world** cloud
  - FEMs, particle methods
  - Amazon EC2 Spot
- Improve **a distributed thread scheduler**
  - Consider the cost of thread migration and data locality
  - Reduce the overhead of running multiple threads on one node
- Support fault tolerance
  - **Distributed checkpointing & restart**

Publications

- A Global Address Space Framework for Irregular Applications (accepted, short paper). High Performance Distributed Computing. 2010/6
- **原健太朗，田浦健次朗，近山隆．**DMI**：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース．情報処理学会論文誌（プログラミング）．**Vol.3，No.1，pp.1-40．2010/3
- **原健太朗．有限要素法における連立方程式ソルバの並列化（第** 2 **回クラスタシステム上の並列プログラミングコンテスト成果報告）．第** 9 **回** PC **クラスタシンポジウム．**2009/12
- **原健太朗，田浦健次朗，近山隆．**DMI**：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース．**SWoPP2009．2009/8