Lightweight Checkpointing of Loop-Based Kernels Using Disaggregated Memory

Jie Li
Ph.D. Student, TTU
02/22/2022

Overview

- Background
- Motivation
- Methodology
- Summary & Future Work

Checkpoint/Restart

- Checkpoint/Restart for **fault tolerance**.
  - Applications running on HPC Systems can encounter Mean-Time-Between-Failures (MTBF) on the order of hours or days due to hardware breakdowns and software errors.
- Checkpoint/Restart for **job migration**.
  - Renewable-energy-powered HPC Systems require jobs to be suspended and migrated to fit within the power availability.
- Checkpointing: periodically **saving applications state to checkpoint files** on reliable storage, typically a parallel file system.
- Restarting: application **restarts from a prior state** by reading in a checkpoint file.

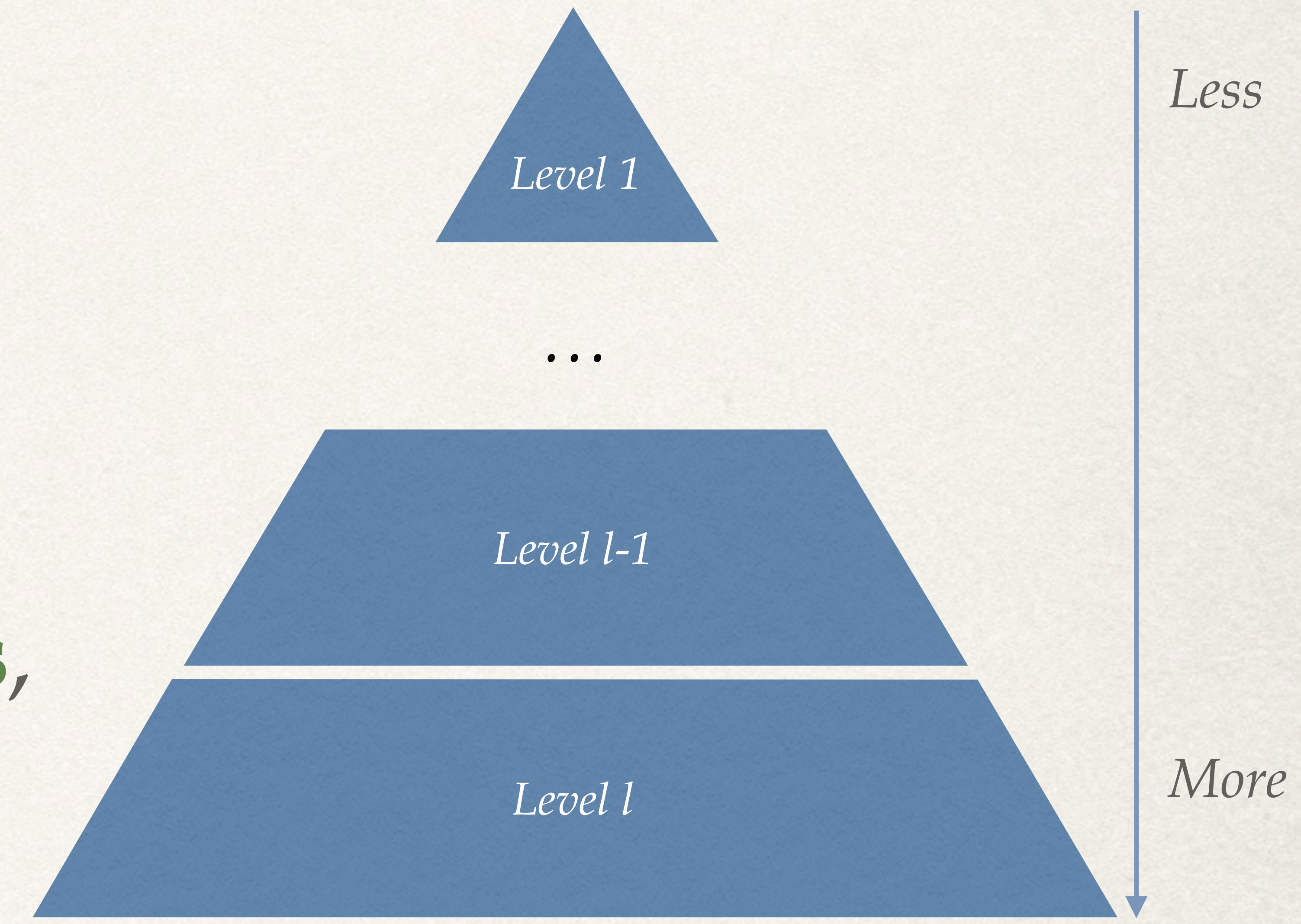
Checkpoint/Restart

	System Level Checkpointing	Application Level Checkpointing
Programmer Effort	No code changes on applications ✓	Applications need to be modified, recompiled or relinked
Portability	Hardly portable	Easier with checkpointing run-time library ✓
Checkpoint Size	Large checkpoint file, may including unnecessary temporary data	Relatively small if users carefully choose the checkpoint region ✓
Flexibility	"Blind" time-triggered checkpoints; Less flexible	Data-driven or iteration-based checkpoint; More flexible ✓
Restartability	Restart is usually restricted to homogeneous hosts	More easily portable and can be restarted on different systems ✓

Silva, Luís Moura, and João Gabriel Silva. "System-level versus user-defined checkpointing." In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281)*, pp. 68-74. IEEE, 1998.

Multi-level Checkpoint

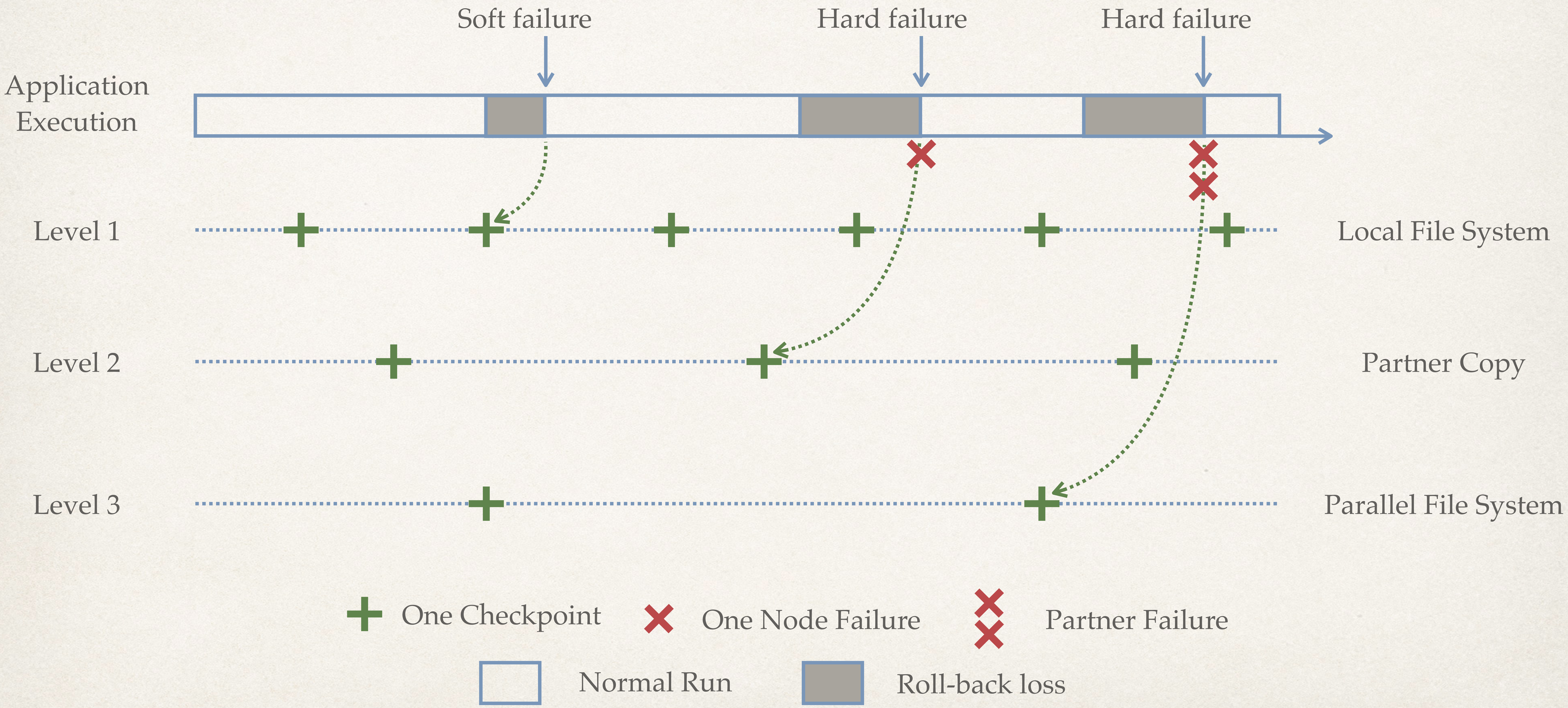
- Traditional Checkpoint/Restart is not suitable for exascale system
  - HPC systems experience significantly **high failure rates**
  - Checkpointing becomes **more critical** but **less practical**
- Multi-level checkpoint
  - The same checkpoint data are stored **using several mechanisms**, each of which have a different cost and level of resilience
  - NOT all failures require costly restarts of the application from a parallel file system
  - Less severe failures can be restarted in significantly less time from higher (faster) levels



Time Cost and Resilience of Checkpoint Levels

Moody, Adam, Greg Bronevetsky, Kathryn Mohror, and Bronis R. De Supinski. "Design, modeling, and evaluation of a scalable multi-level checkpointing system." In SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-11. IEEE, 2010.

Multi-level Checkpoint



Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications, https://slideplayer.com/slide/7871177/

Motivation

- HPC Systems often use diskless compute nodes
  - Node local disk is a scarce resource
- Only storage available on some systems is RAM disc
  - RAM disc consumes main memory
- Restarting from a different node relies on parallel file systems
  - High overhead and overwhelms file system resources

A Lightweight Checkpoint/Restart Layer

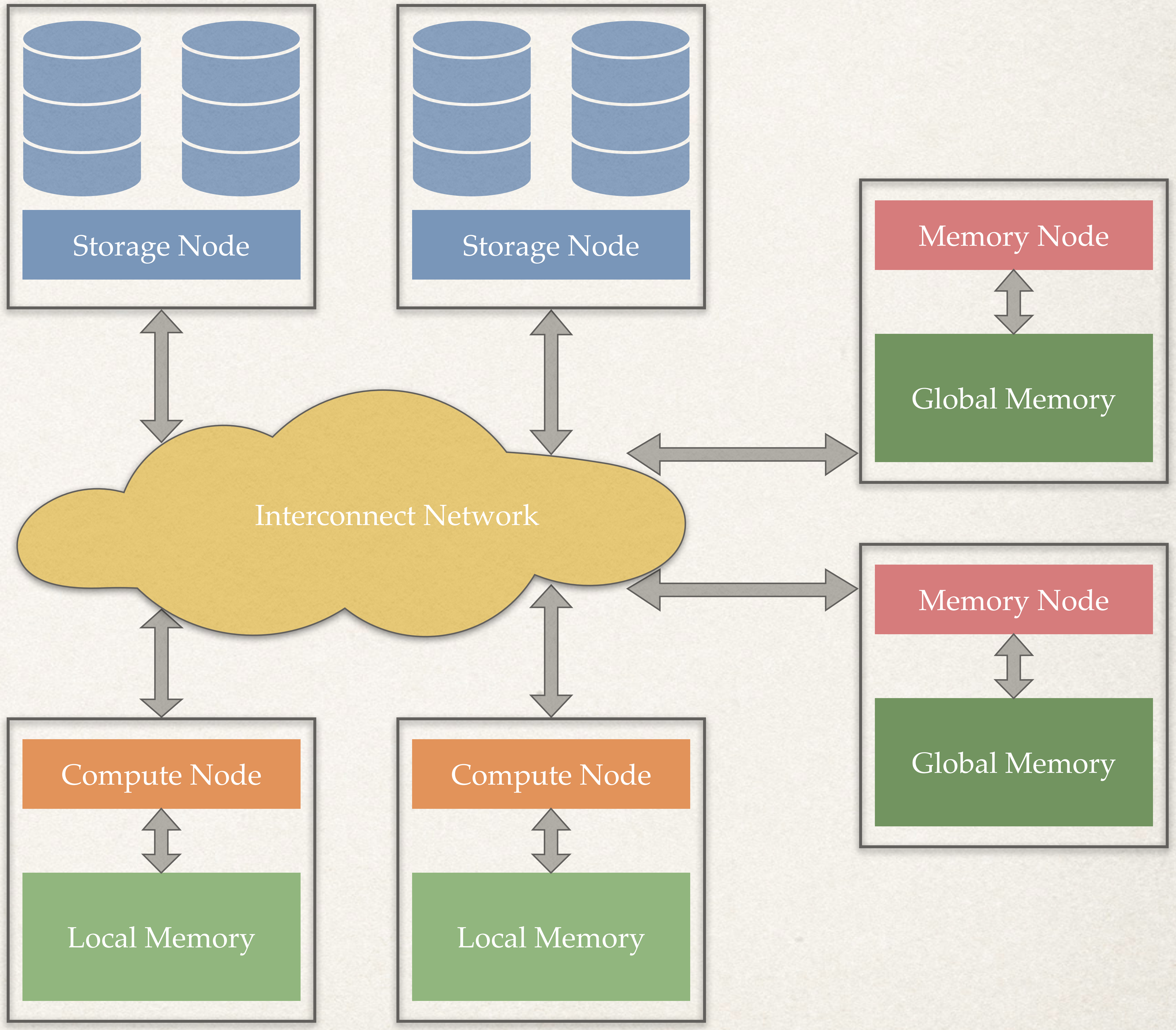
Does NOT consume resources of compute nodes

Reduces reliance on parallel file systems when restarting from hard failures

Rajachandrasekar, Raghunath, Adam Moody, Kathryn Mohror, and Dhabaleswar K. Panda. "A 1 PB/s file system to checkpoint three million MPI tasks." In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pp. 143-154. 2013.

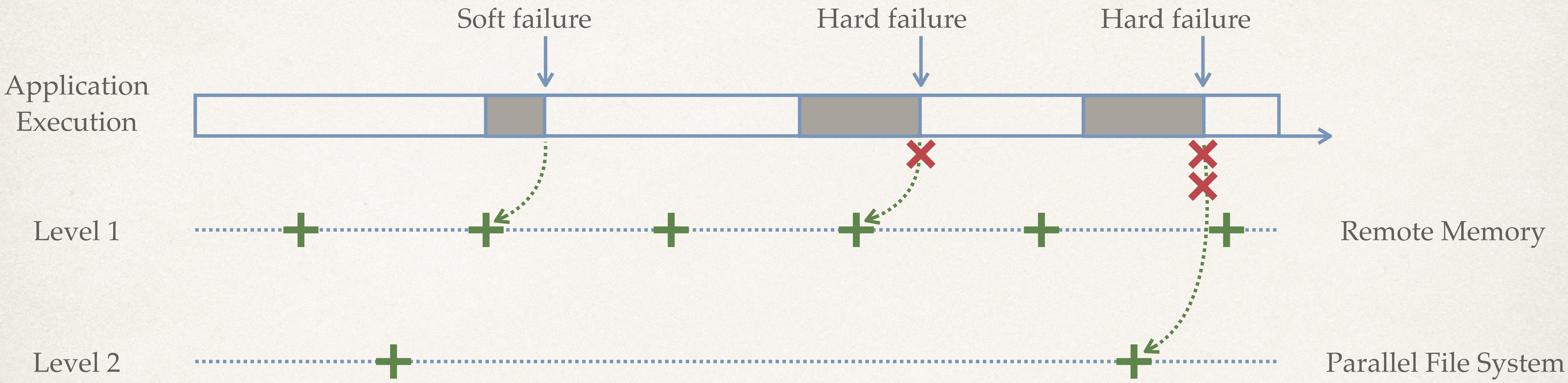
Disaggregated Memory based on xBGAS

- **Disaggregated Memory**: memory, as a critical resource, can be disaggregated from compute nodes via xBGAS (Extended Global Address Space).
- Memory nodes serve as **a global-shared memory pool** that provides a large memory capacity for compute nodes.
- A process's memory resources are **not limited to its local memory**, but extend to the remote shared memory pool.
- The latencies of memory and network are converging; there will be **comparable cost** to access local memory or remote memory.

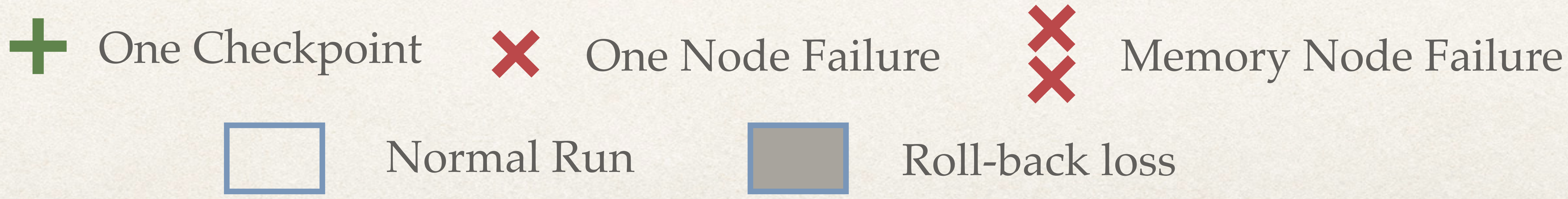


HPC System Architecture with the Disaggregated Memory

Two-level Checkpoint with Disaggregated Memory



Use Disaggregated (remote) memory as the C/R target



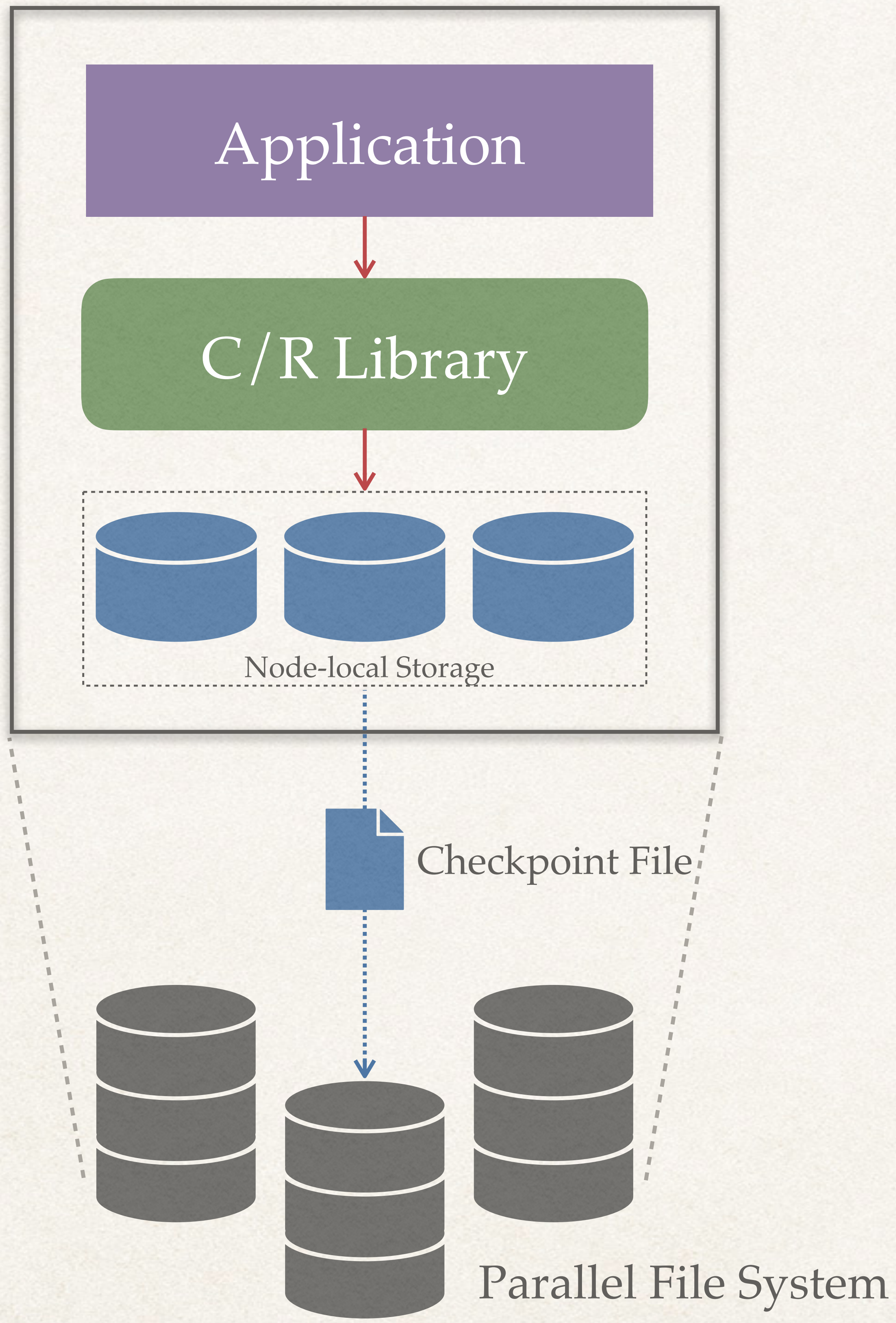
Challenges and Opportunities

- Current popular C/R techniques (BLCR, SCR, DMTCP, etc.) save system/application state in files.
  - Implement an in-memory file system on remote memory?
  - This is doable (and a different project), but it might be **overkill** for C/R.
- Observations:
  - Most HPC applications have certain **key data structures** or **variables** from which the programmer counter and processes' stack can be **restored**.
  - There is **not need to store the whole state of a process**, reducing the amount of checkpoint data copied to remote memory.
- Opportunities:
  - The proposed C/R **only saves the critical data** of applications **on remote memory**.
  - Checkpoint data is stored in memory and **can be updated in place**.

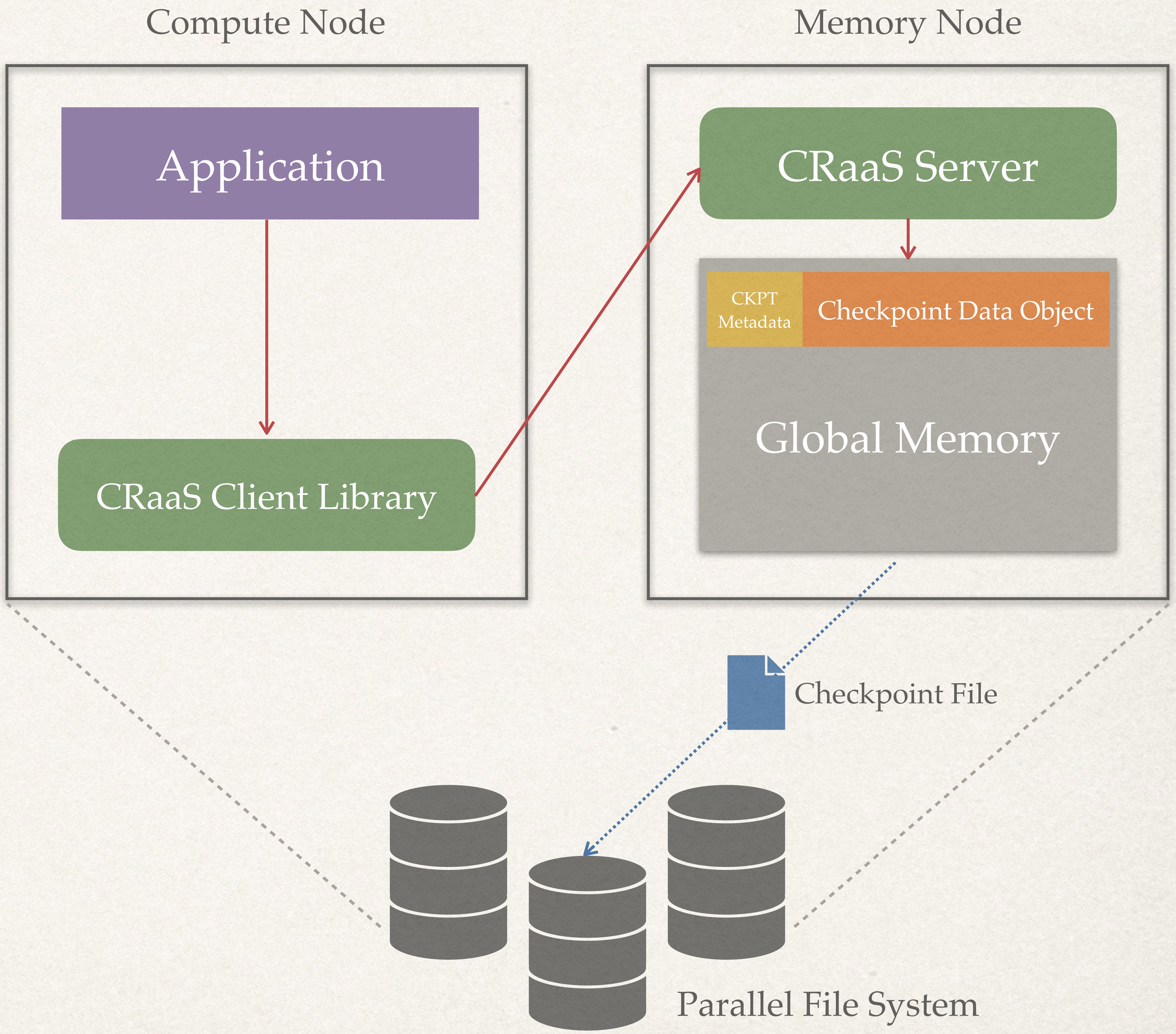
Design Philosophy

- **C/R as a service** (CRaaS): C/R is provided as a service for all applications running on HPC Systems.
- Checkpoint/Restart is performed through **CRaaS APIs**, which take advantages of the low-overhead **xBGAS** to transfer checkpoint data.
- CRaaS only guarantees the **integrity** of the checkpoint data. It is up to users to specify which data should be checkpointed and when/where to do the checkpoint.
- CRaaS provides **redundancy** of critical data on remote memory by periodically (and less frequently) writing to parallel file systems.

CRaaS Architecture



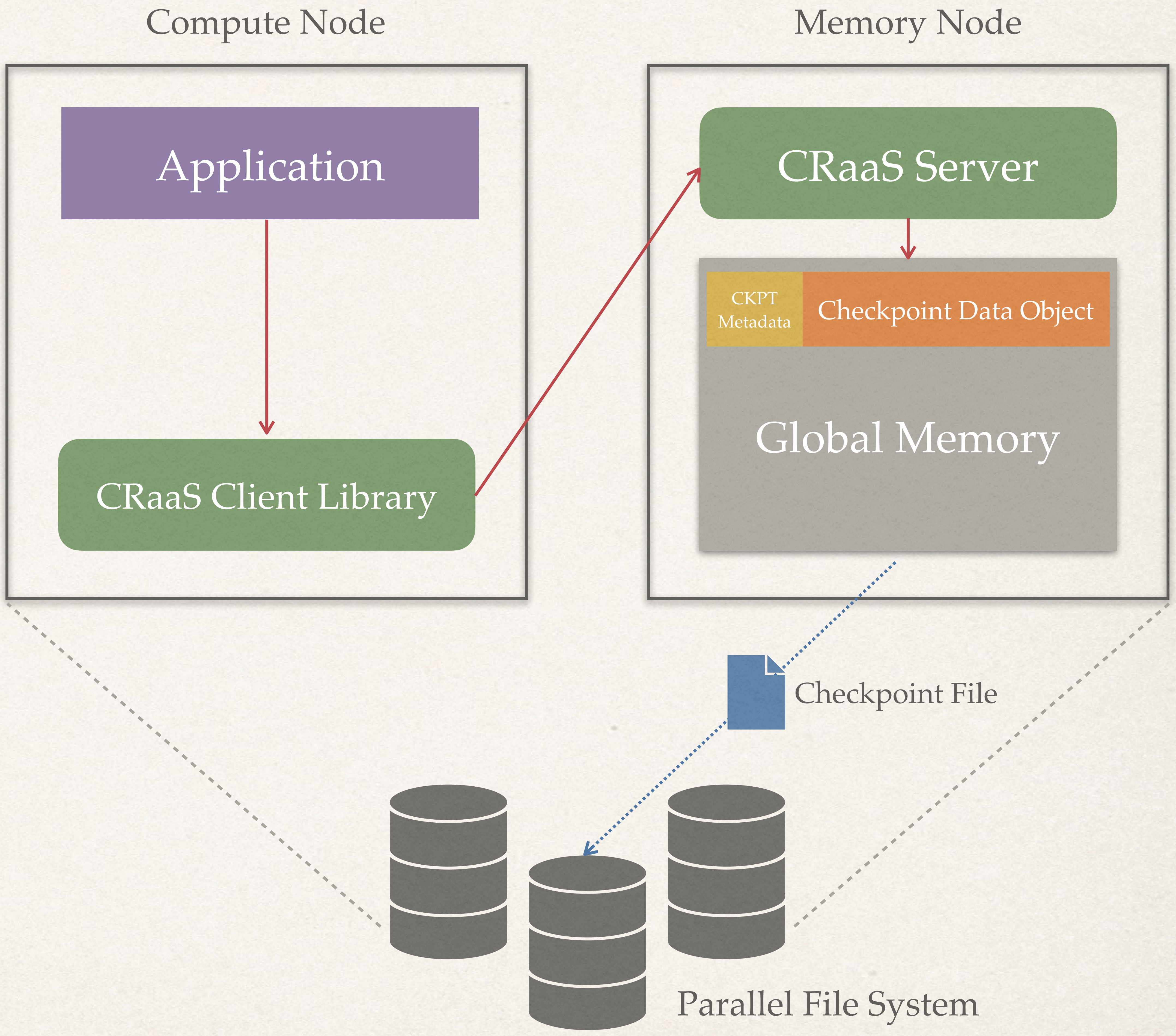
Architecture of Two-Level C/R



Architecture of CRaaS

CRaaS Architecture

- Checkpoint:
  - Checkpoints are directly to *CRaaS* and saved on the allocated remote memory region (level-1).
  - *CRaaS* on memory node periodically flushes checkpoints to parallel file system for persistent storage (level-2).
- Restart:
  - *CRaaS* first checks if the application's checkpoint exists in remote memory, and if so, sends the checkpoint back to the application.
  - If not, *CRaaS* reads the corresponding file from parallel file system, allocates memory for storing the checkpoint, and copies back to the application.



Architecture of CRaaS

Loop-based Kernels

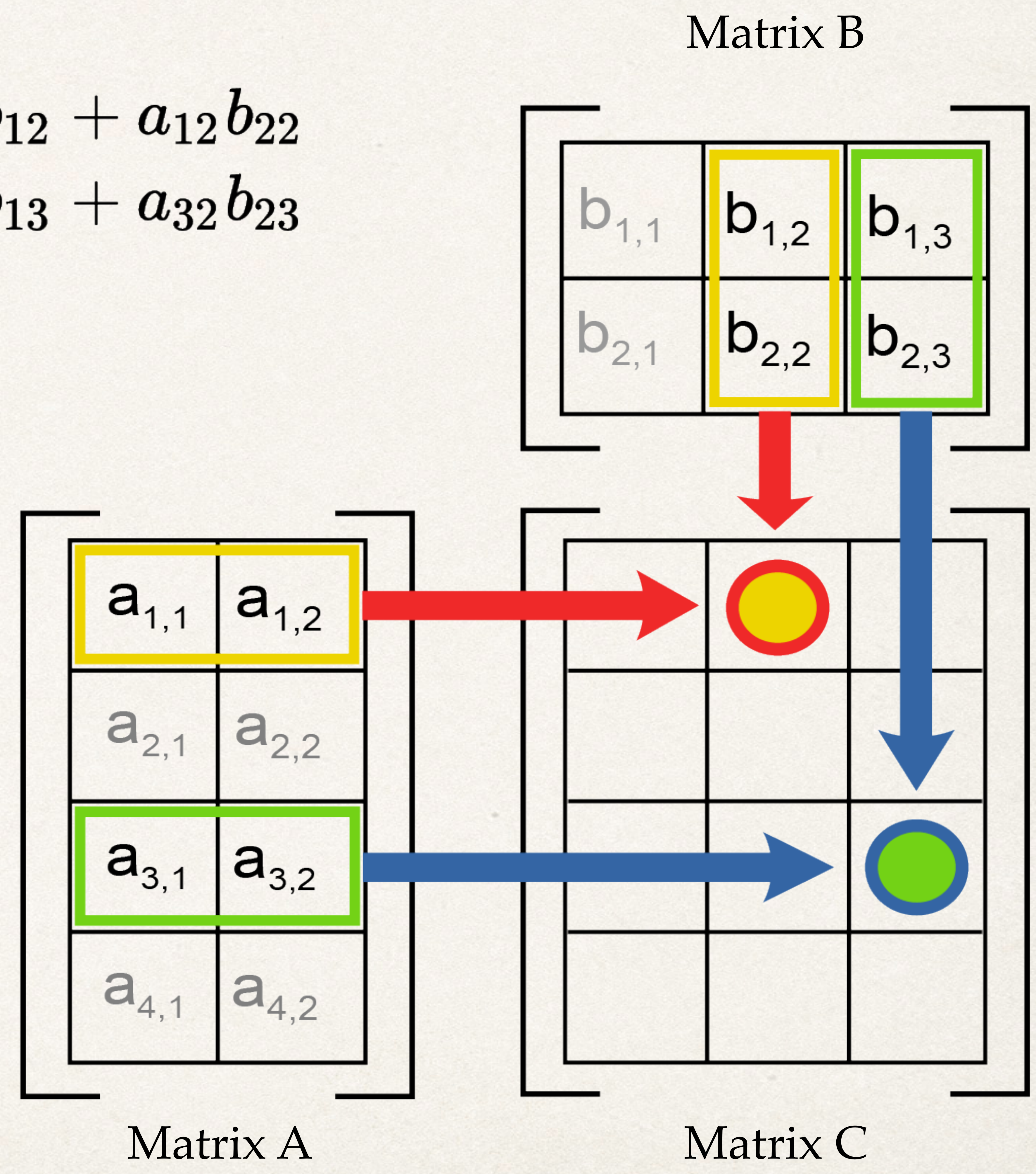
- **Matrix Multiplication**: Frequent and heavy-duty mathematical operation in machine learning.
- LU decomposition: solving linear equations, inverting a metric, computing the determinant.
- Fast Fourier Transform: digital recording, sampling, additive synthesis and pitch correction software.
- Gaussian elimination: computing determinants, finding the inverse of a matrix, computing ranks and bases.
- …

https://en.wikipedia.org/wiki/Matrix_multiplication; https://en.wikipedia.org/wiki/LU_decomposition; https://en.wikipedia.org/wiki/Fast_Fourier_transform; https://en.wikipedia.org/wiki/Gaussian_elimination

Matrix Multiplication

$$\underset{4\times 2\text{ matrix}}{\begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix}} \underset{2\times 3\text{ matrix}}{\begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix}} = \underset{4\times 3\text{ matrix}}{\begin{bmatrix} \cdot & c_{12} & c_{13} \\ \cdot & \cdot & \cdot \\ \cdot & c_{32} & c_{33} \\ \cdot & \cdot & \cdot \end{bmatrix}}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{33} = a_{31}b_{13} + a_{32}b_{23}$$

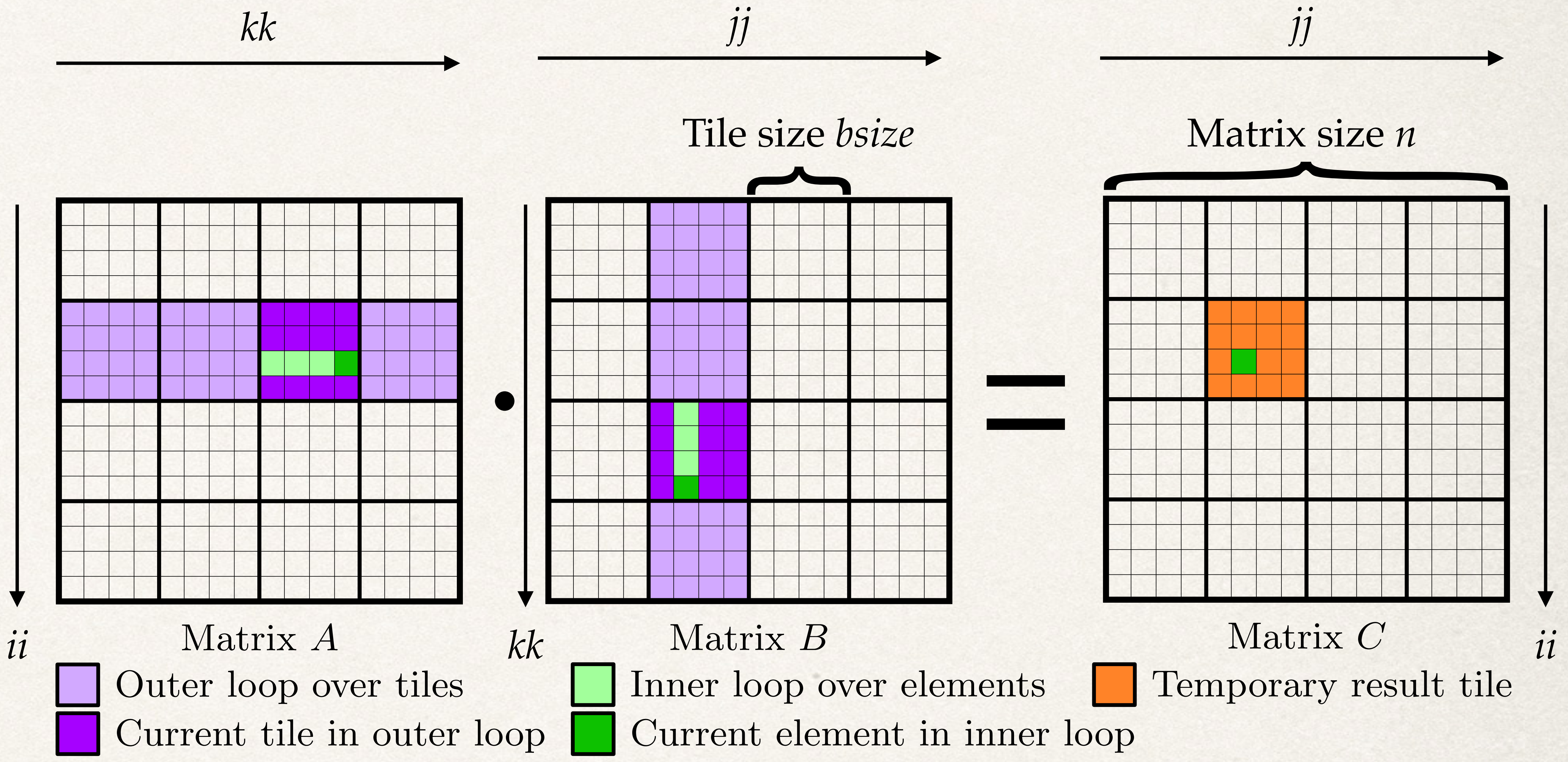


A: m x n, B: n x p, C: m x p

```
for (i=0; i<m; i++) {
    for (j=0; j<p; j++)
        for(k=0; k<n; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

Tiled Matrix Multiplication (TMM)


```
for (kk=0; kk<n; kk+=bsize) { // move horizontally on Matrix A, and vertically on Metric B
    for (ii=0; ii<n; ii+=bsize) { // move vertically on Matrix A and C
        for (jj=0; jj<n; jj+=bsize) { // move horizontally on Matrix B and  C
            for (i=ii; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum=c[i][j];
                    for(k=kk; k<(kk+bsize); k++) {
                        sum+=a[i][k]*b[k][j];
                    } // end of k
                    c[i][j]=sum;
                } // end of j
            } // end of i
        } // end of jj
    } // end of ii
} // end of kk
```



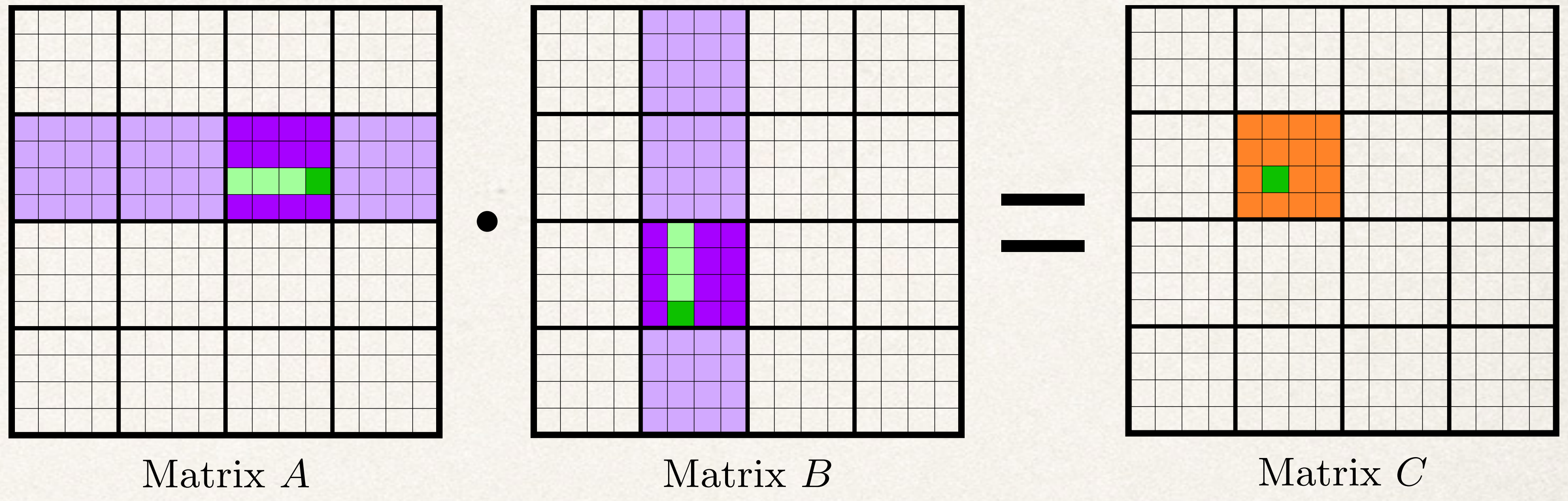
6-loop tiling for matrix-matrix multiplication

Elnawawy, Hussein, Mohammad Alshboul, James Tuck, and Yan Solihin. "Efficient checkpointing of loop-based codes for non-volatile main memory." In 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 318-329. IEEE, 2017.

Matthes, Alexander, René Widera, Erik Zenker, Benjamin Worpitz, Axel Huebl, and Michael Bussmann. "Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library." In *International Conference on High Performance Computing*, pp. 496-514. Springer, Cham, 2017.

Critical Data in TMM

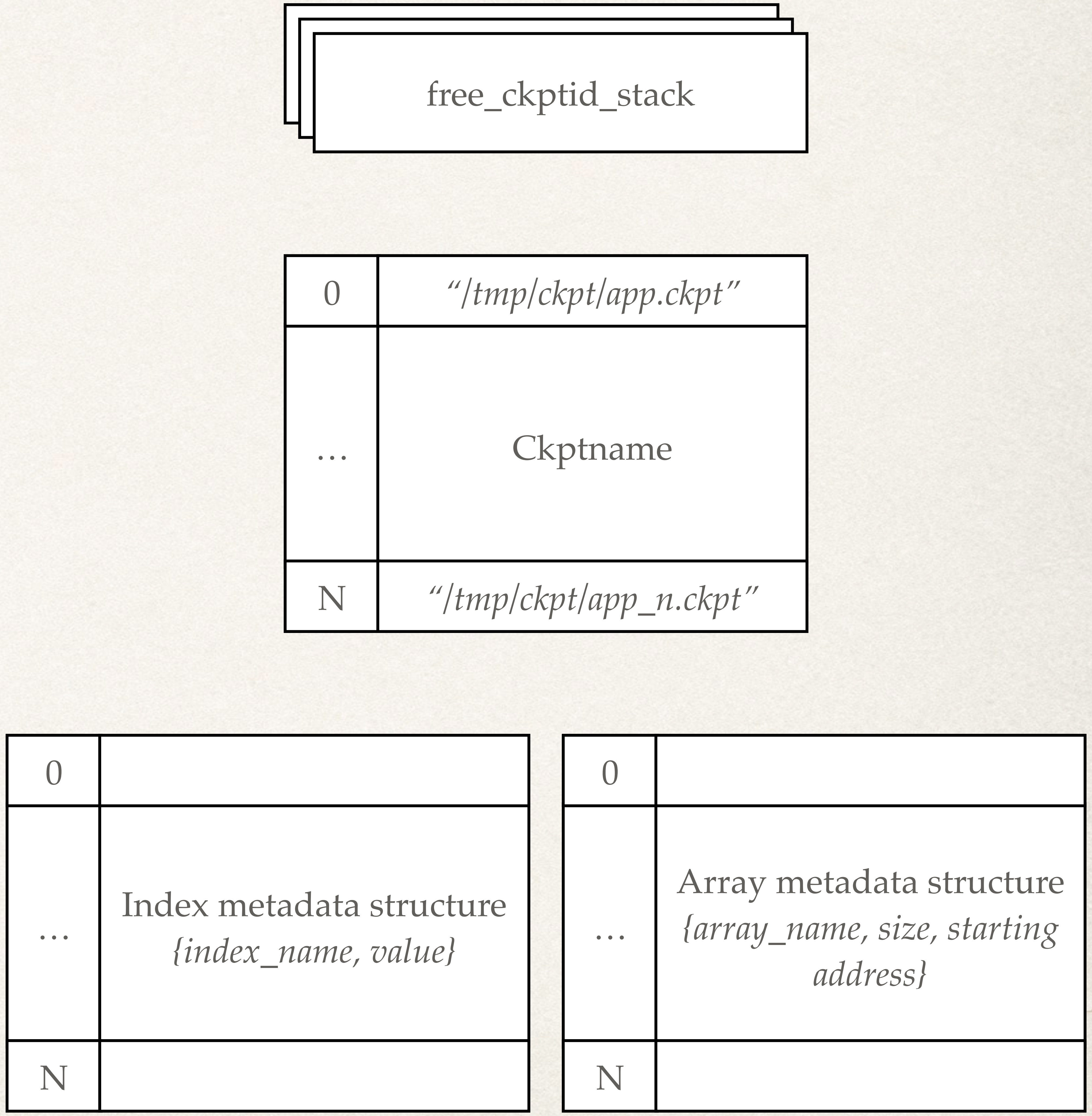
- Checkpointing the work that was already completed:
  - **The elements of matrix C**: only matrix C is updated during the computing
  - **Loop indices** for determining where in the loop-nest to resume execution in recovery
- We do not have to checkpoint all indices in the loops; some updates on matrix C can be discarded on failures.
  - Coarse granularity of checkpointing reduces the checkpointing frequency with a cost of recomputing some elements in matrix C.
- Upon failure, we recover a state by reading the indices and determining where in the loop-nest to resume execution.




```
for (kk=0; kk<n; kk+=bsize) {
    for (ii=0; ii<n; ii+=bsize) {
        for (jj=0; jj<n; jj+=bsize) {
            for (i=ii; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum=c[i][j];
                    for(k=kk; k<(kk+bsize); k++) {
                        sum+=a[i][k]*b[k][j];
                    }
                    c[i][j]=sum;
. . .
```

CRaaS Data Structure

- Ckptname: the path to the checkpoint file on the parallel file system, e.g, *"/tmp/ckpt/app.ckpt"*.
  - To create the checkpoint file on the parallel file system,
  - To find the corresponding entry of the checkpoint data object on the remote memory.
- CkptID: each checkpoint is assigned an internal Checkpoint ID value, which is managed by *free_ckptid_stack*.
  - When a new checkpoint is created, *CRaaS* pops the next available CkptID from the stack.
  - When a checkpoint is deleted, its associated CkptID is pushed back onto the stack.
- CkptID-Ckptname table: records the mapping between CkptID and Ckptname.
- Indices metadata table, matrix metadata table. CkptID is used as an index into these two tables.



CRaaS APIs Design

- CRaaS_Init(char *ckptname) *# Initialize the CRaaS client; check the availability of the CRaaS service; add metadata entries if CkptID is not found; create a checkpoint file is no checkpoint file exists.*
- CRaaS_Finalize(char *ckptname) *# The application has finished successfully. Close the CRaaS client; delete the metadata entries; release the allocated remote memory region.*
- APIs for checkpointing/restarting the data objects:
  - CRaaS_Ckpt_index(char *ckptname, “index_name”, index_value) *# Store index value*
  - CRaaS_Ckpt_array(char *ckptname, “array_name”, array, size_of_array) *# Store array*
  - CRaaS_Rst_index(char *ckptname, “index_name”) *# Load index value*
  - CRaaS_Rst_array(char *ckptname, “array_name”, array, size_of_array) *# Load array*

Note that we do not have to distinguish between a normal start and a recover restart

CRaaS on TMM


```
last_kk = CRaaS_Rst_index(ckptname, “kk”);
CRaaS_Rst_array(ckptname, “c”, c, n*n*sizeof(int));
for (kk= last_kk ; kk<n; kk+=bsize) {
    for (ii=0; ii<n; ii+=bsize) {
        for (jj=0; jj<n; jj+=bsize) {
            for (i=ii; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum=c[i][j];
                    for(k=kk; k<(kk+bsize); k++) {
                        sum+=a[i][k]*b[k][j];
                    }
                    c[i][j]=sum;
        . . .
        } // end of jj
    } // end of ii
    CRaaS_Ckpt_index(ckptname, “kk”, kk+bsize);
    CRaaS_Ckpt_array(ckptname, “c”, c, n * n * sizeof(int));
}
```

C/R on *kk* loop

```
last_ii = CRaaS_Rst_index(ckptname, “ii”);
last_kk = CRaaS_Rst_index(ckptname, “kk”);
CRaaS_Rst_array(ckptname, “c”, c, n*n*sizeof(int));
for (kk= last_kk ; kk<n; kk+=bsize) {
    for (ii= last_ii ; ii<n; ii+=bsize) {
        for (jj=0; jj<n; jj+=bsize) {
            for (i=ii; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum=c[i][j];
                    for(k=kk; k<(kk+bsize); k++) {
                        sum+=a[i][k]*b[k][j];
                    }
                    c[i][j]=sum;
        . . .
        } // end of jj
        CRaaS_Ckpt_index(ckptname, “ii”, ii+bsize);
        CRaaS_Ckpt_array(ckptname, “c”, c, n * n * sizeof(int));
    } // end of ii
    CRaaS_Ckpt_index(ckptname, “kk”, kk+bsize);
}
```

C/R on *ii* loop

Potential Issues

- What if the process crashes at the middle of checkpointing? e.g., ii has been stored, but matrix c is not stored.
  - Need a mechanism to guarantee that the checkpointing of matrix and loop indices is **atomic**: all updates are stored, or none are done.
- Checkpointing all elements of matrix c is not necessary since not all elements are updated in the checkpointing loop.
  - Need a smart way to recognize which part of the matrix should be checkpointed.

```
last_ii = CRaaS_Rst_index(ckptname, “ii”);
last_kk = CRaaS_Rst_index(ckptname, “kk”);
CRaaS_Rst_array(ckptname, “c”, c, n*n*sizeof(int));
for (kk= last_kk ; kk<n; kk+=bsize) {
    for (ii= last_ii ; ii<n; ii+=bsize) {
        for (jj=0; jj<n; jj+=bsize) {
            for (i=ii; i<(ii+bsize); i++) {
                for (j=jj; j<(jj+bsize); j++) {
                    sum=c[i][j];
                    for(k=kk; k<(kk+bsize); k++) {
                        sum+=a[i][k]*b[k][j];
                    }
                    c[i][j]=sum;
        . . .
        } // end of jj
        CRaaS_Ckpt_index(ckptname, “ii”, ii+bsize);
        CRaaS_Ckpt_array(ckptname, “c”, c, n * n * sizeof(int));
    } // end of ii
    CRaaS_Ckpt_index(ckptname, “kk”, kk+bsize);
}
```

C/R on *ii* loop

Summary & Future Work

- Presented a **lightweight** C/R technique that checkpoints **critical data objects** of loop-based kernels on **disaggregated memory** (i.e., remote memory).
- Introduced the architecture of proposed C/R, **CRaaS (C/R as s Service)**, and explained its operation mechanism.
- Designed several **data structures** to manage the checkpoint data and defined several **key APIs** to perform checkpointing and restarting.
- Future work:
  - Resolve the potential issues.
  - Integrate with MPI to provide distributed checkpointing.

Thank you!

Q&A