COMP90025 Parallel and Multicore Computing

Parallel architectures and APIs

Aaron Harwood

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

Parallel architectures

The PRAM model for parallel computation is algorithmically powerful and a direct implementation of a PRAM architecture is therefore desirable.
A criticism of the PRAM model is that in practice it is difficult to implement a machine with a potentially unbounded number of processors, that has constant access times to a shared memory across all processors.
In this section we consider possible architectural solutions for parallelism and the additional complexity that they exhibit. We can then revisit our models and provide refinements that include the architectural complexities.

PRAM considerations

The PRAM model considers N processors and M memory locations which are sometimes called *shared variables*.
While each processor may have its private memory, i.e. registers or cache, these are not shared with other processors.
A PRAM implementation must specify how the processors and memory locations are arranged so that the PRAM model is not violated, i.e. in a constant time step each of the processors can, in parallel, access a random memory location.

Flynn's classifications of parallel architectures

Flynn identifies four architectural classifications when considering the implementation of parallelism:

SISD : single instruction single data,

SIMD : single instruction multiple data,

MISD : multiple instruction single data,

MIMD : multiple instruction multiple data,

The definitions consider two fundamental streams, the *instruction* stream and the *data* stream.

All of the above parallelism can be simulated by a PRAM. The PRAM acts like a synchronous MIMD. But these architectures do not tell us how memory and processors are organized.

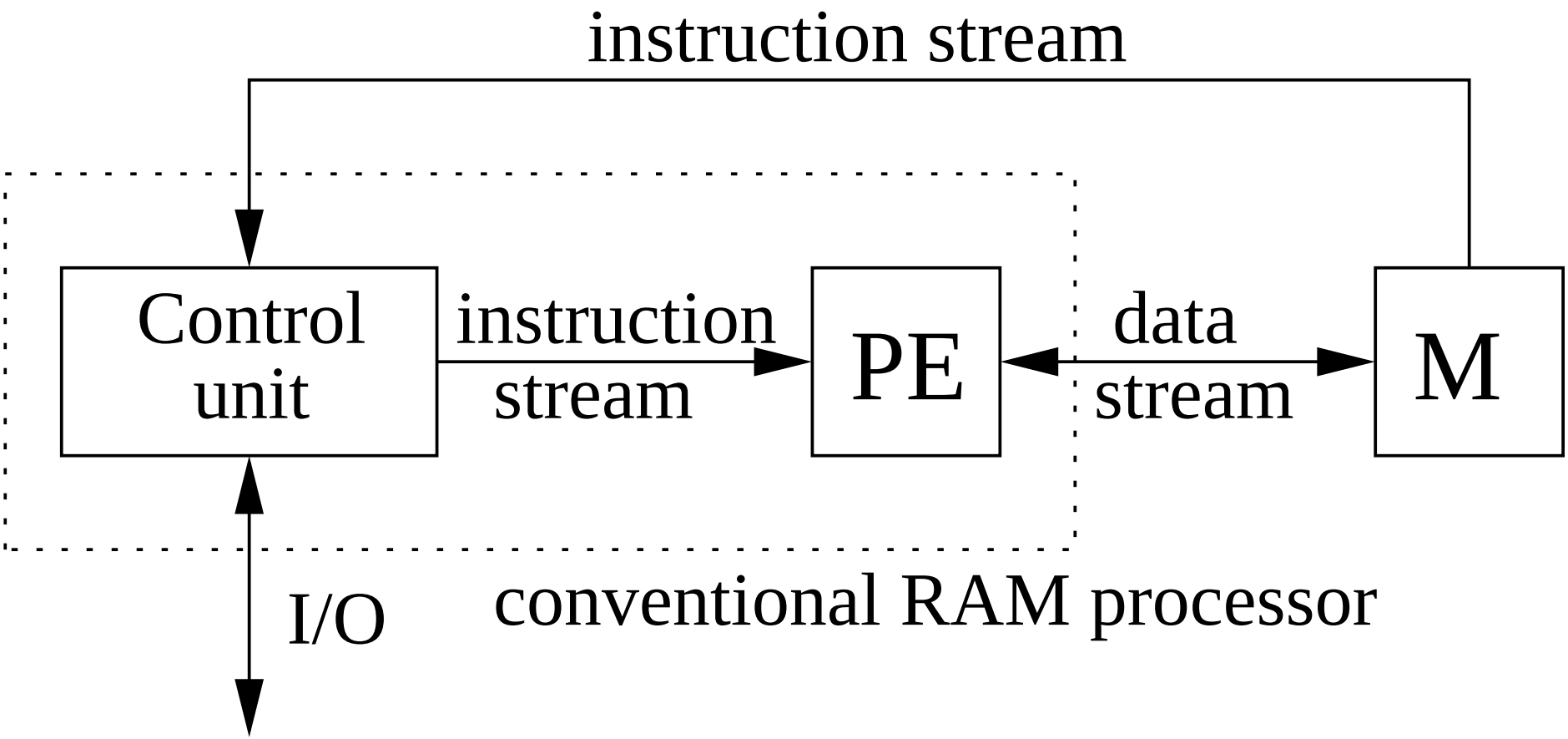

Figure: Abstract SISD

SIMD machine

For applications with lots of *data parallelism*, the most cost effective platforms are SIMD machines. In these machines, a single control unit broadcasts (micro-) instructions to many *processing elements* (each of which is a set of functional units with local storage) in parallel.
Until the rise of GPUs, the best known SIMD computer was the Connection Machine from Thinking Machines. The CM-2 model had 64k PEs, and even though each PE is only four bits wide, the machine could outperform many big Crays on some specially programmed problems.

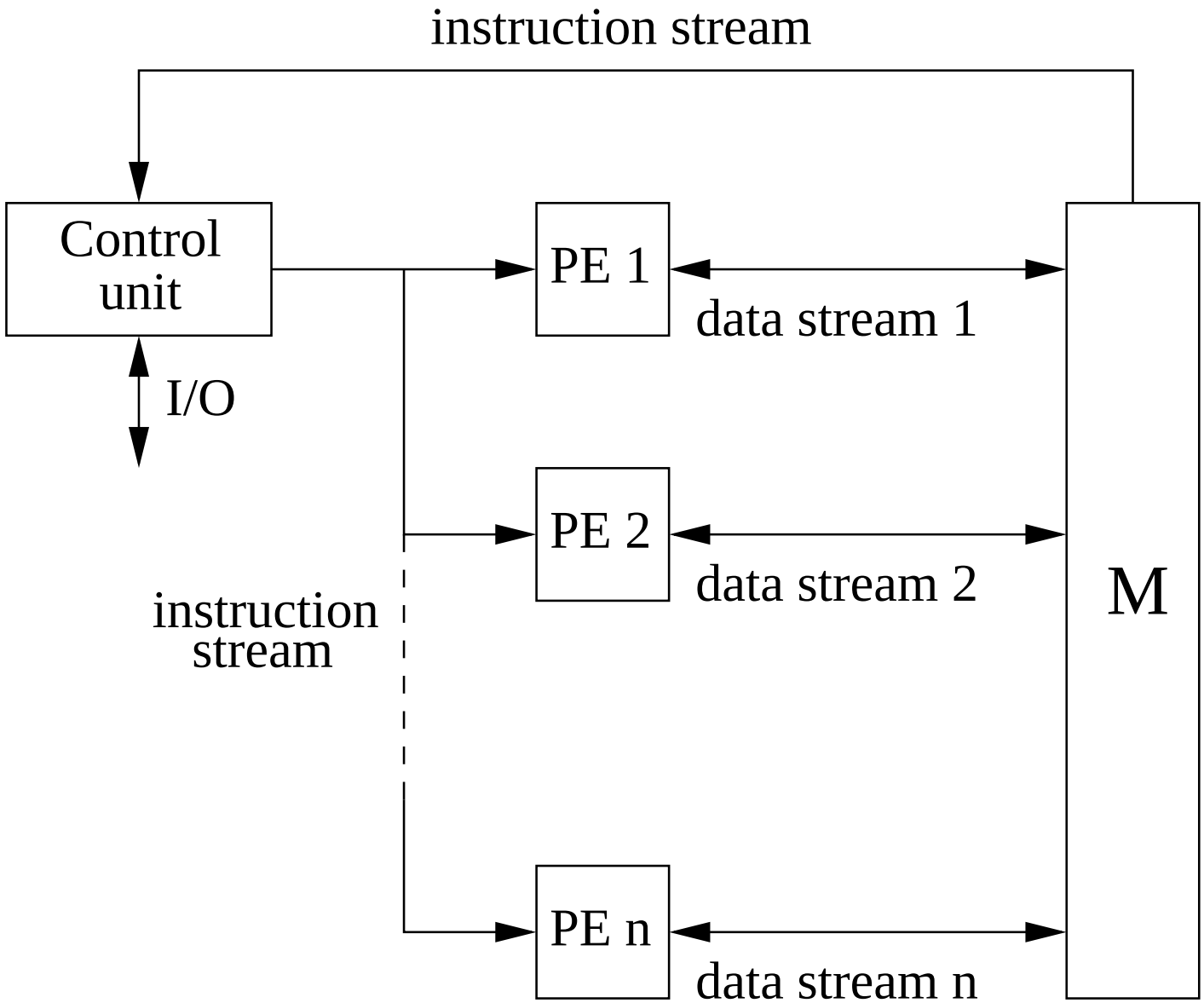

Figure: Abstract SIMD

MIMD machine

Most multiprocessors on the market today are shared memory MIMD machines. They are built out of standard processors and standard memory chips, interconnected by a fast bus (memory is interleaved).

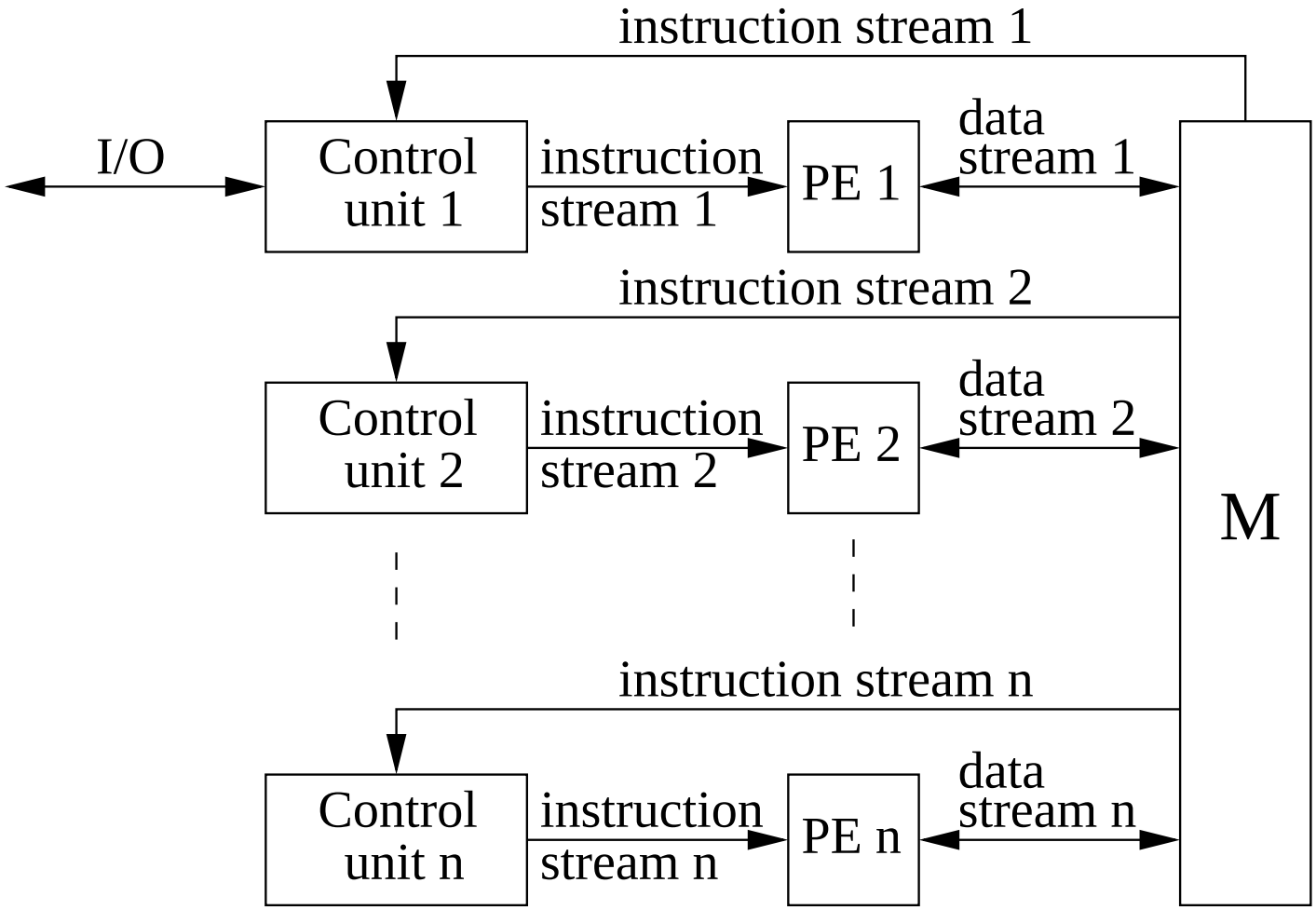

Figure: Abstract MIMD

Schwartz's parallel machine classes

Schwartz defines two general approaches to arranging processors and memory.

Paracomputers separate the memory from the processors. The memory is shared and processors communicate via the shared memory. The PRAM module is closely approximated as a paracomputer in the sense that each memory location is equally accessible to each processor.

Ultracomputers distribute the memory over the processors (leading to modules). A processor can access the memory on its module in constant time but accessing memory on remote modules can take longer.

Alternate terminology is

- paracomputers = *shared memory multiprocessor*
- ultracomputer = *distributed memory multiprocessor*

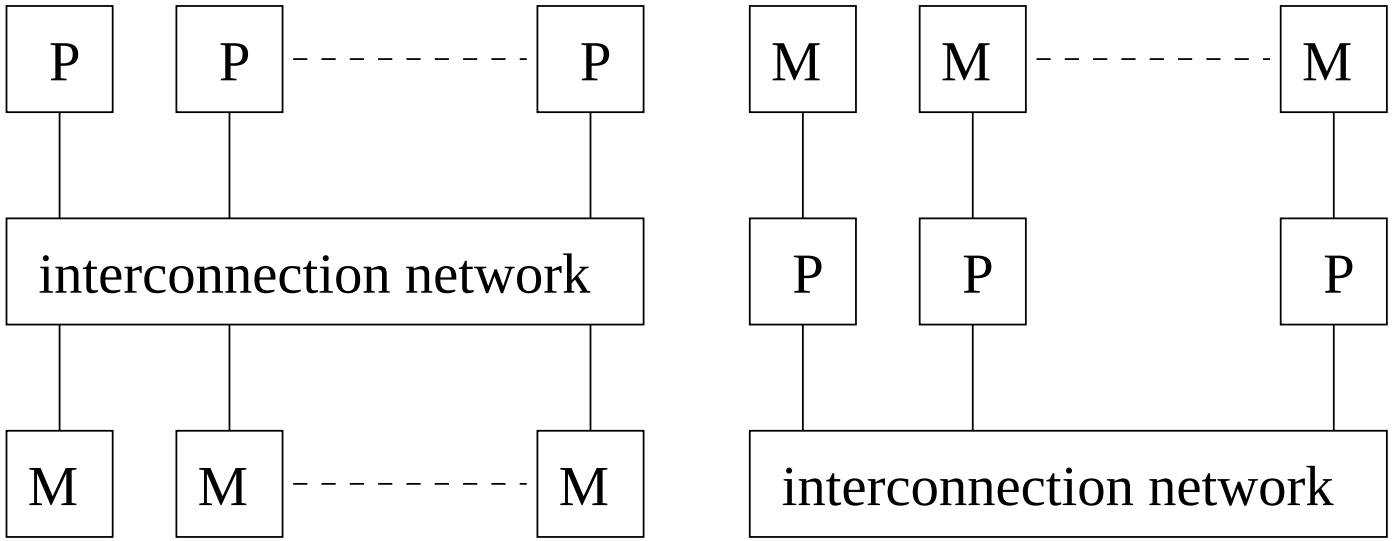

Figure: Shared (left) and distributed memory (right).

Shared versus distributed memory

In a shared memory system, processors communicate by reading and writing messages to a globally known memory address. The hardware ensures that all processors have the same access to the memory, i.e. using a *single address space*. For this reason it is often called a *symmetric multiprocessor* (SMP) machine.

In a distributed memory system, processors communicate by sending messages to each other. The hardware is only responsible for delivering the messages. There is no single address space. For this reason it is often called a *message passing* machine.

A *distributed shared memory* (DSM) system has distributed memory but a single address space.

Programming an SMP machine is easier than programming a message passing machine. However a message passing machine can scale up with less cost.

Uniformity of shared memory access

Depending on the interconnection network, a shared memory machine can be either:

- UMA : uniform memory access - all processors have equal access time to any given memory location.
- NUMA : non-uniform memory access - typically exhibited by DSM architecture, memory locations incur different access delays depending on which processor accesses them.
- COMA : the memory consists only of the collective cache contents of the processors and data migrates to the requesting processor.

UMA and cache issues

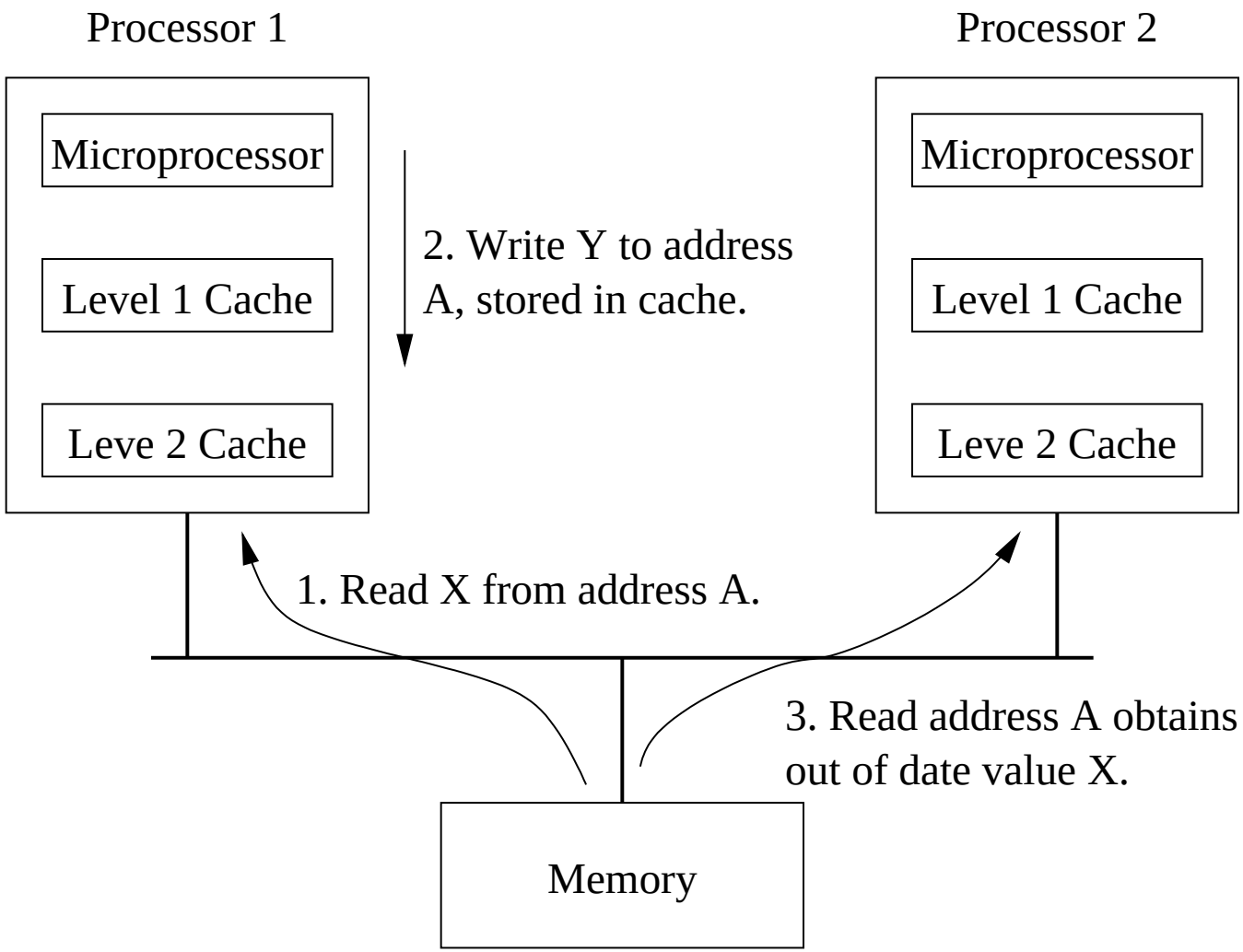

Figure: Caches improve performance but multi-processor machines need protocols to keep cache coherency.

NUMA

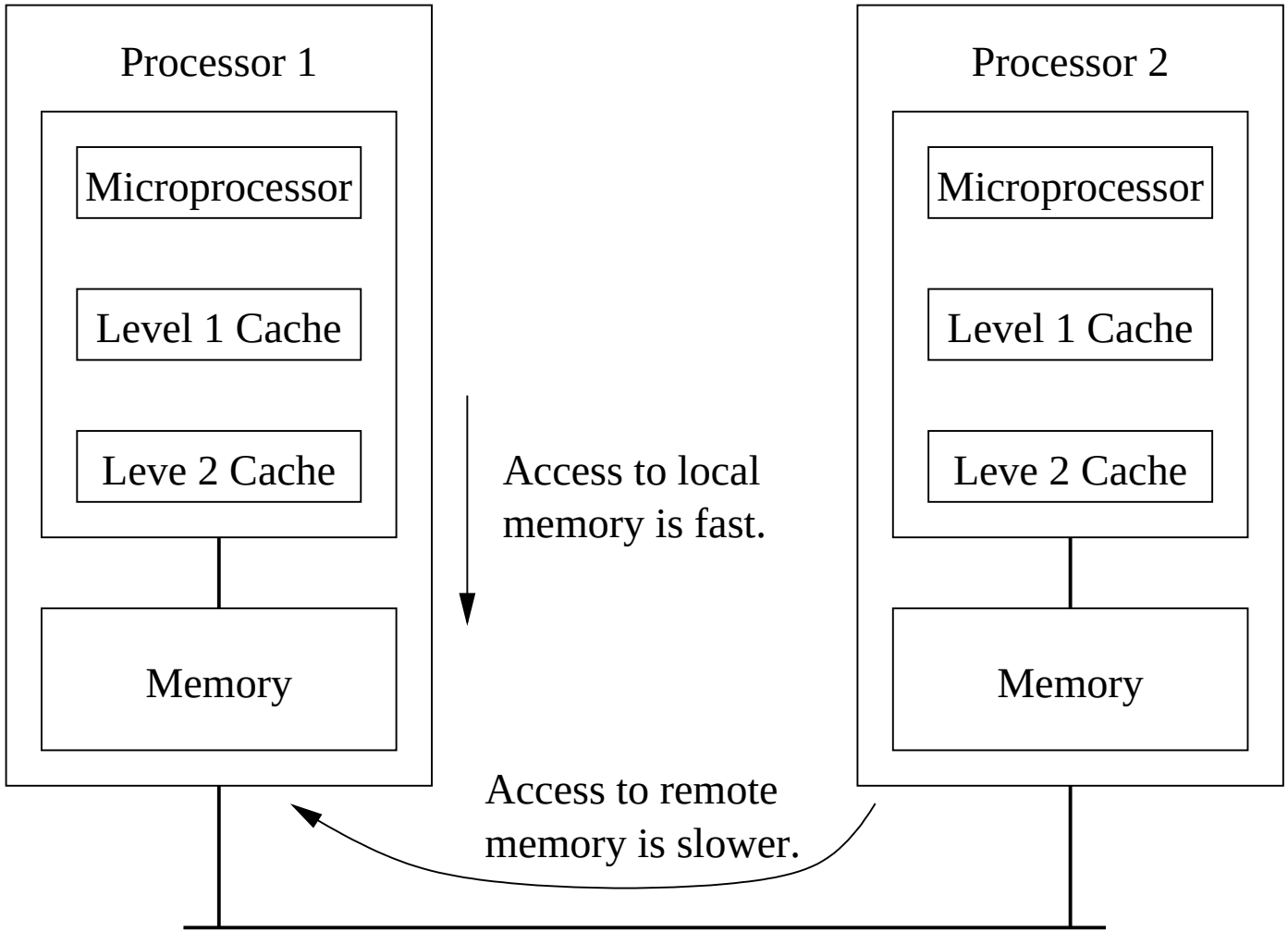

Figure: Distributed memory leads to non-uniform memory access. Cache problems exist for single address space systems.

Simultaneous memory access

Recall the abstract models of memory:

- EREW (exclusive-read-exclusive-write
- CRCW (concurrent-read-concurrent-write) etc.

In these, access to one address is independent of access to any other.
Real memory is none of these.
Memory has pages (virtual memory) and cache (fast copies of recently used memory).
Memory also comes in *modules*.

- There is a long delay between requesting data and receiving it
- After a read, a module must "recharge" before it can be read again

Addressing memory banks

These limitations affect which addresses each memory bank should contain

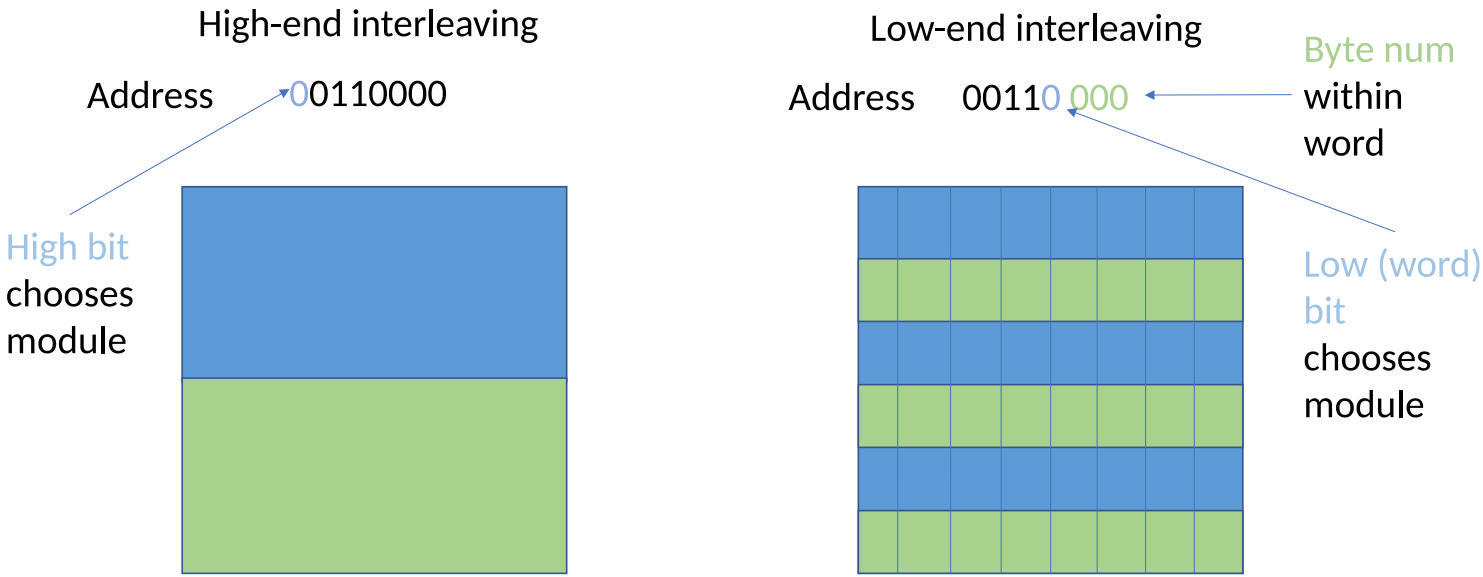

Figure: High-end (left) and low-end (right) interleaving

Coprocessors

Coprocessors are common these days, e.g. GPUs and Xeon Phi. They provide very low cost parallelism but architecturally they present more challenges.

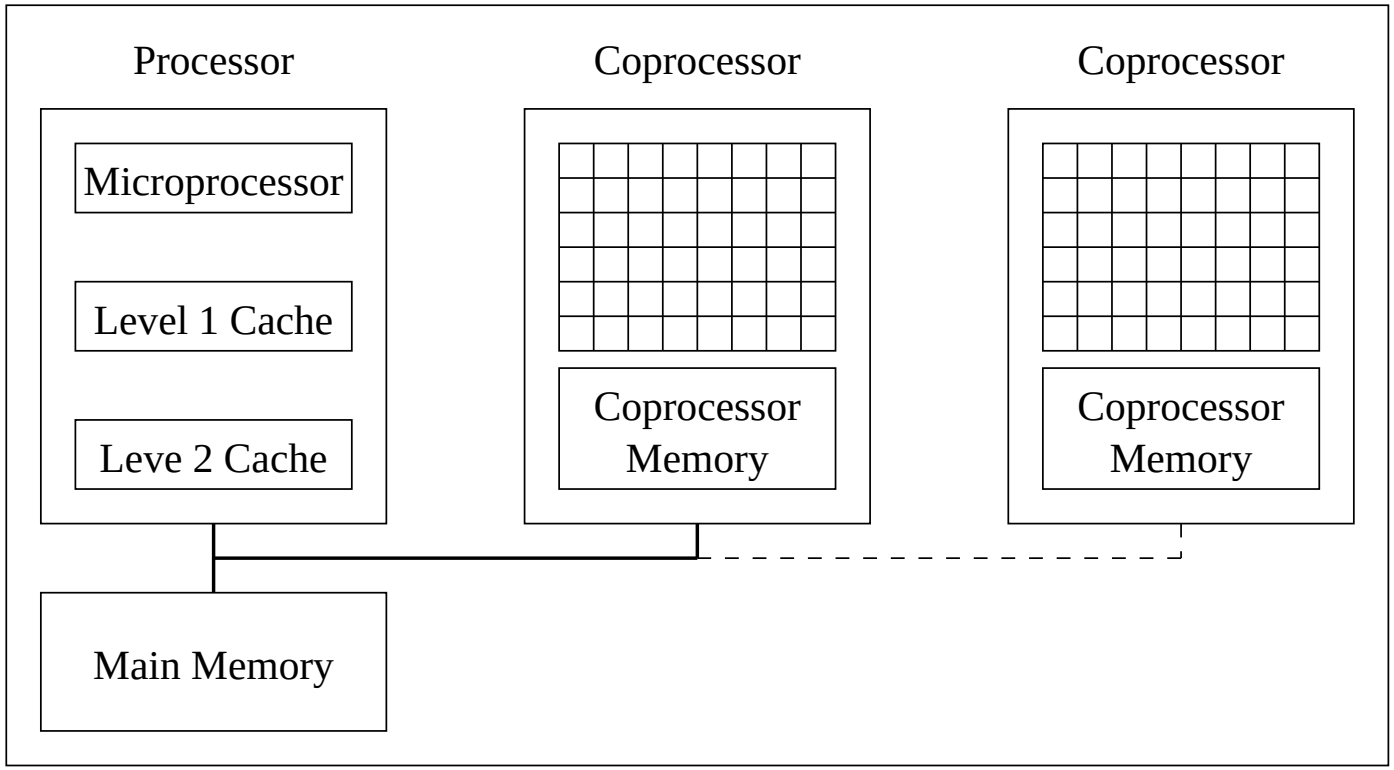

Figure: The coprocessor has its own memory and processing elements. Programs and data need to be transferred to and from the main memory and coprocessor memory.

Implicit versus Explicit

Models of parallel processing can be considered in a number of different ways (Skillicorn, *ACM Computing Survey*, Vol. 30, No. 2, 1998):

parallelism implicit/explicit: is a description of the parallel algorithm explicitly required?

decomposition implicit/explicit: are the pieces of the parallel program explicitly defined?

mapping implicit/explicit: are the pieces of the parallel program required to be explicitly mapped to different processors?

communication implicit/explicit: is the communication between pieces of the parallel program required to be explicitly defined?

Threaded model for parallelism

The threaded model for parallelism is suitable when running on a single machine, i.e. single process with a single address space, or a distributed shared memory architecture that implicitly distributes threads (very rare). Note that coprocessors are usually treated as a separate machine, although this is increasingly being blurred with new developments (e.g., Java transparently running on GPUs).

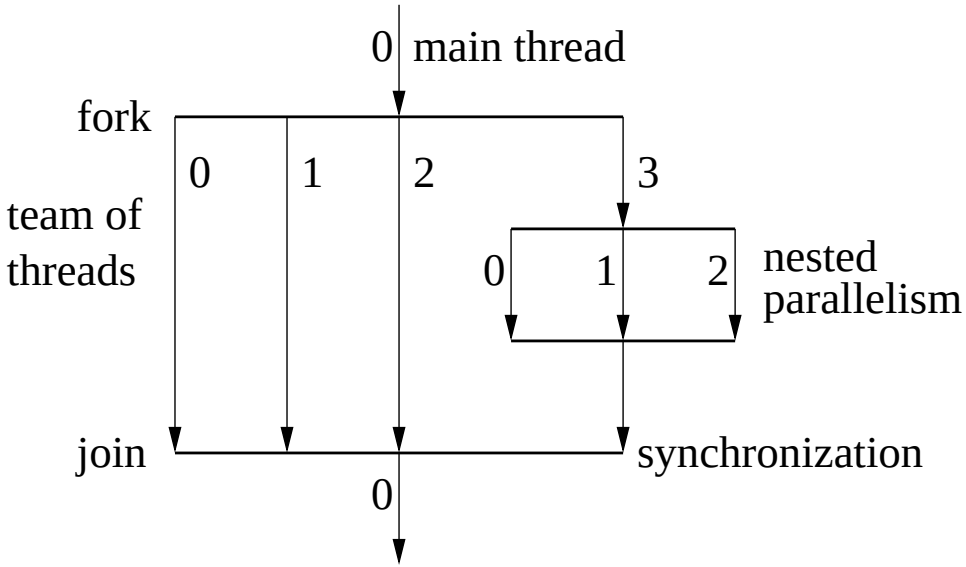

Figure: All processes start with a single, main thred, and create new threads as needed.

Process model for parallelism

The process model for parallelism is suitable when running on multiple machines, i.e. when there is no shared memory. Again, usually the process model is not transparent to coprocessors. The task model usually requires explicit decomposition and communication. Sometimes we use the word *task* to mean a process in a multi-process program.

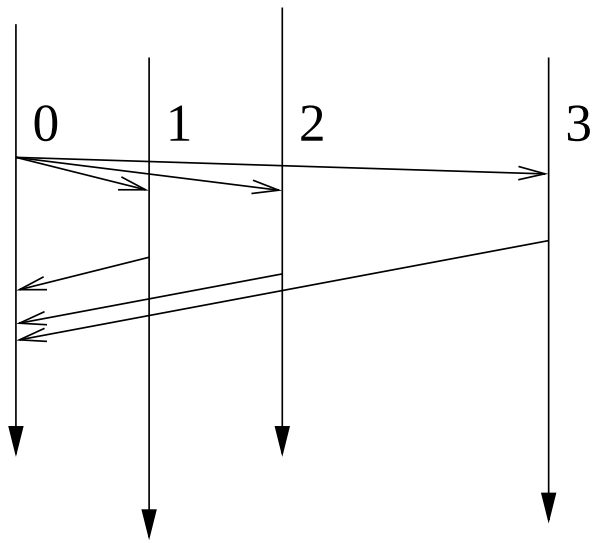

Figure: Processes run on individual machines and communicate by sending messages to each other. Some global knowledge, such as process number is usually available.

Hybrid model for parallelism

The hybrid model involves both multiple processes and multiple threads. Making use of a coprocessor can be considered a hybrid model in most cases.

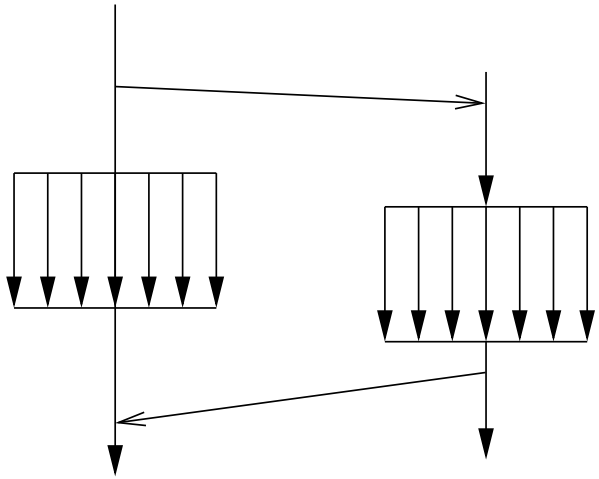

Figure: Two processes with each process making use of threads internally.

Since the thread model is more efficienct than the process model on a multicore machine, a distributed memory architecture where each machine has multiple cores would use the hybrid model, running a single process on each machine and having each process use multiple threads.

OpenMP

OpenMP, or Open Multi-Processing, is an API that supports shared memory multiprocessing programming in C, and other languages, across a wide range of platform. It is parallelism-explicit and mostly implicit in all other ways.
E.g. each iteration of this loop is assigned to a thread pool:

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Java 1.8+ parallelism

While OpenMP is the mainstay in utilizing a multicore machine, Java 1.8 and above provides "parallelism" (its actually more akin to concurrency), in much the same way, via streams and lambda functions:

```
int[] a = new int[100000];

IntStream.range(0, 100000)
    .parallel()
    .forEach(i ->
        {
            a[i]=2*i;
        }
);
```

Similarly to OpenMP, this is parallelism-explicit and mostly implicit in all other ways.

omp4j

The *omp4j* project, `http://www.omp4j.org/`, provides OpenMP for Java, using an *omp4j* compiler rather than *javac*.

```
int[] a = new int[100000];

// omp parallel for
for (int i = 0; i < 100000; i++) {
    a[i]=2*i;
}
```

In this case the comment before the loop tells *omp4j* that the loop should be parallelized.

OpenMPI

OpenMPI, or Open Message Passing Interface, is for distributed memory machines where parallelism, decomposition and message passing are explicitly defined. Mapping can sometimes be defined but is usually implicit. E.g., usually the first process (master or process id 0) sends data to the other processes and receives processed data back:

```
if(my_id == 0) { // master
  for(an_id = 1; an_id < num_procs; an_id++) {
    start_row = an_id*num_rows_per_process;
    MPI_Send( &num_rows_to_send, 1, MPI_INT,
      an_id, send_data_tag, MPI_COMM_WORLD);
    MPI_Send( &array1[start_row], num_rows_per_process,
      MPI_FLOAT, an_id, send_data_tag, MPI_COMM_WORLD);
  for(an_id = 1; an_id < num_procs; an_id++) {
    MPI_Recv( &array2, num_rows_returned, MPI_FLOAT,
      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
} else { // slave process

}
```

OpenCL

Open Computing Language is a framework for writing programs across heterogeneous platforms, e.g. consisting of multicore processors and coprocessors (GPUs, digital signal processors, etc).

```
// Multiplies A*x, leaving the result in y.
// A(i,j) is at A[i*ncols+j].
__kernel void matvec(__global const float *A,
                     __global const float *x,
                     uint ncols, __global float *y)
{
    size_t i = get_global_id(0);
    __global float const *a = &A[i*ncols];
    float sum = 0.f;
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

SPMD and MPMD

The terms *Single Program Multiple Data* (SPMD) and *Multiple Program Multiple Data* (MPMD) in this subject will be used to categorize the way in which the parallel program is written, not the architecture, i.e. do not become confused with Flynn's Classifications.
For example, usually an OpenMP or OpenMPI program is written as a single program, and each instance (task or thread) of the program, operates on a different portion of the data, determined by its task identifier or rank. This is a SPMD approach.
Whereas in MPMD, different programs are written, e.g. perhaps in OpenMPI there is a separate program for the master as to the slave tasks. Still the same division of data is determined at the slaves by their task or rank identifier. Or indeed the MPMD approach may consist of many different programs, perhaps in a pipeline fashion, to achieve parallelism.