# Analytical Evaluation of Parallel Performance

Serial code: $T_{total}(1) = T_{setup} + T_{compute} + T_{finalize}$

Parallel with P processes
Gain perfect speed on compute section

Parallel code: $T_{total}(P) = T_{setup} + T_{compute}(1)/P + T_{finalize}$

Relative speedup: $S(P) = T_{total}(1)/T_{total}(p)$

Serial fraction: $\gamma = (T_{setup} + T_{finalize}) / T_{total}(1)$

# Amdahl's Law
# (Fixed-Size Speedup)

↓ Rewrite $T_{total}(P)$ with $\gamma$

$$T_{total}(P) = \gamma T_{total}(1) + (1-\gamma)T_{total}(1)/P$$

↓ Rewrite $S(P)$ with $T_{total}(P)$

Amdahl's Law: $$S(P) = T_{total}(1) / (\gamma T_{total}(1) + (1-\gamma)T_{total}(1)/P)$$
$$= 1/(\gamma + (1-\gamma)/P)$$

# Upper Bound on Speedup

If P -> infinite, S(P) = 1/$\gamma$ -- upper bound

One use case:

If $\gamma = 20\%$, S(P) = 5 regardless of the number of processes

A Climate Model:

$T_{total} = T_{dynamics} + T_{radiation} + T_{moisture\ etc} + T_{IO}$

= 25% + 20% + 35 % + 20%

= 80% (parallel)+ 20% (serial)

# Performance Improvement Is Limited by Algorithms

Assume 8x speedup (e.g., replace Intel Woodcrest with IBM Cell), that is P=8, then S=3.333 (not 1/0.2 = 5)

16x speedup, that is P=16, then S=4.0 (not 2x3.333)

----> Further performance improvement needs to change algorithms. For example, convert a serial IO into a parallel IO

# Assumptions in Amdahl's Law

- Creating additional parallel tasks may increase overhead (?) and the chances of contentions (?) for shared resources
  - Longer running time
- Increasing the number of processors may improve the performance serial portion (?)
  - Shorter running time
- A serial algorithm may perform less number of computational steps (?)
  - Shorter running time

# Gustafson's Law (Fixed-Time Speedup)

Serial code: $T_{total}(1) = T_{setup} + T_{compute} + T_{finalize}$

Rewrite for the case of P processors

$$T_{total}(1) = T_{setup} + PT_{compute}(P) + T_{finalize}$$

Scaled serial fraction: $\gamma_{scaled} = (T_{setup} + T_{finalize}) / T_{total}(P)$

$$T_{total}(1) = \gamma_{scaled}\, T_{total}(P) + P(1- \gamma_{scaled})T_{total}(P)$$

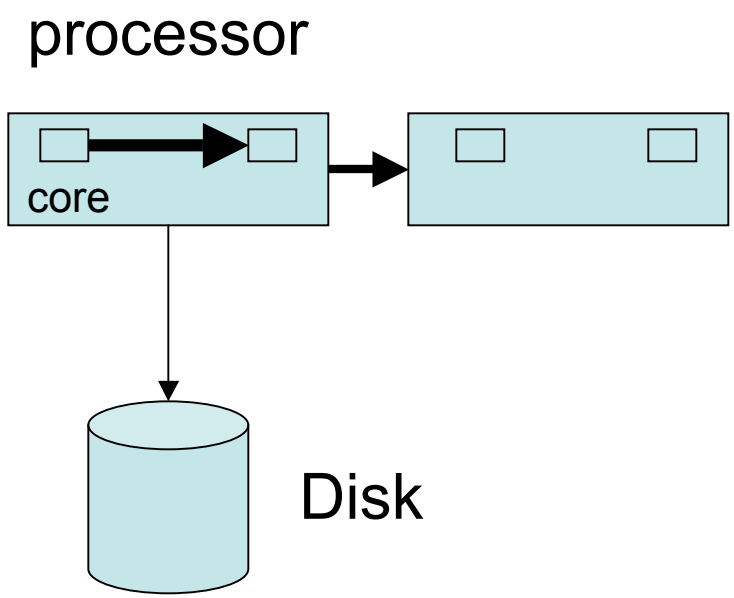
Scaled (or fixed-time) speed up: $S(P) = P + (1- P)\, \gamma_{scaled}$

# Latency and Bandwidth

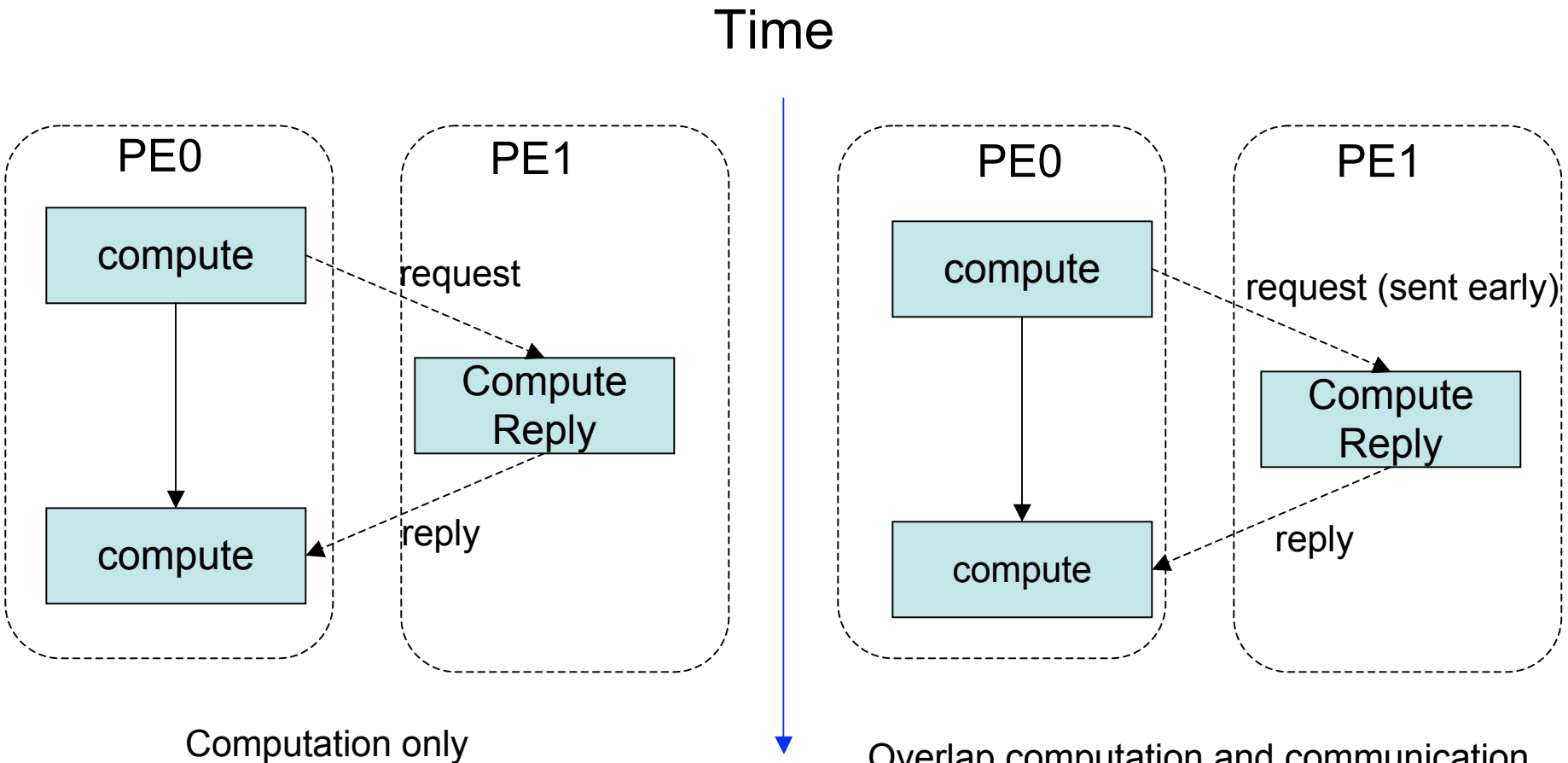
- $T_{message\text{-}transfer} = \alpha + N/\beta$
  - Latency, $\alpha$, is a fixed cost
    - The time taken to send an empty message over the communication medium, from the time the send routine is called and to the time the data is received by the recipient.
    - Include overhead due to software and network hardware plus the time for the message to traverse the communication medium
    - Unit: time unit
  - Bandwidth, $\beta$, is a measure of the capacity of the communication medium
    - Unit: bytes per time unit
  - Message length, N
    - Unit: byte

# Optimize Performance Based on Latency and Bandwidth

- For a system with a large latency, it is worthwhile to restructure a problem that sends many small messages to aggregate the communication into a few large messages instead
  - Send data across processors which interconnected through the network
  - Send data to disks



# Overlapping Communication and Computation and Latency Hiding



# Optimization - Double Buffering

IBM Cell's SPE uses DMA to move data and instruction between main storage and the local store (LS) in the SPE.

A simple scheme to achieve that data transfer:

1. Start a DMA data transfer from main storage to buffer B0 in the LS
2. Wait for the transfer to complete
3. Use the data in buffer B0
4. Repeat

The purpose of double buffering is to maximize the time spent in the compute phase of a program and minimize the time spent waiting for DMA transfers to complete.

# Optimization - Double Buffering

Global Circulation Model  Main Memory  Data Transfer via DMA  Local Store

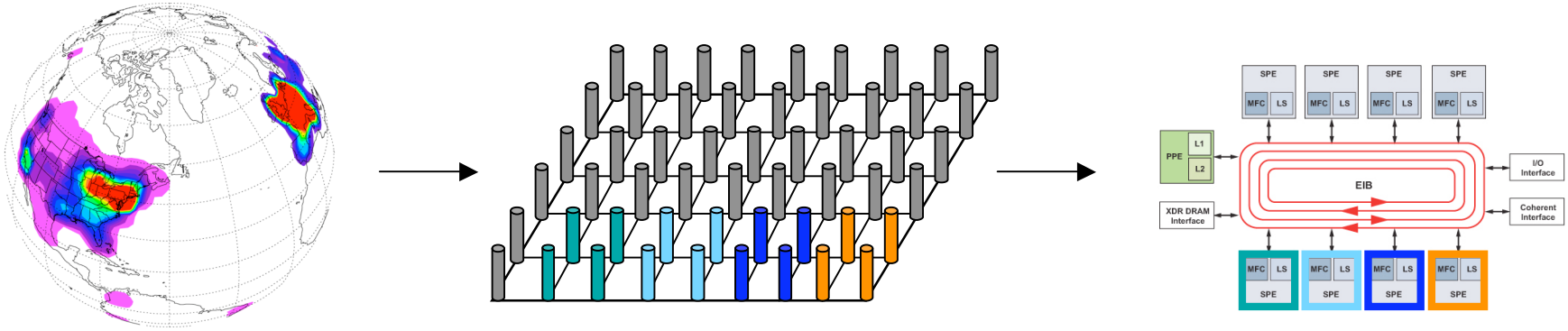
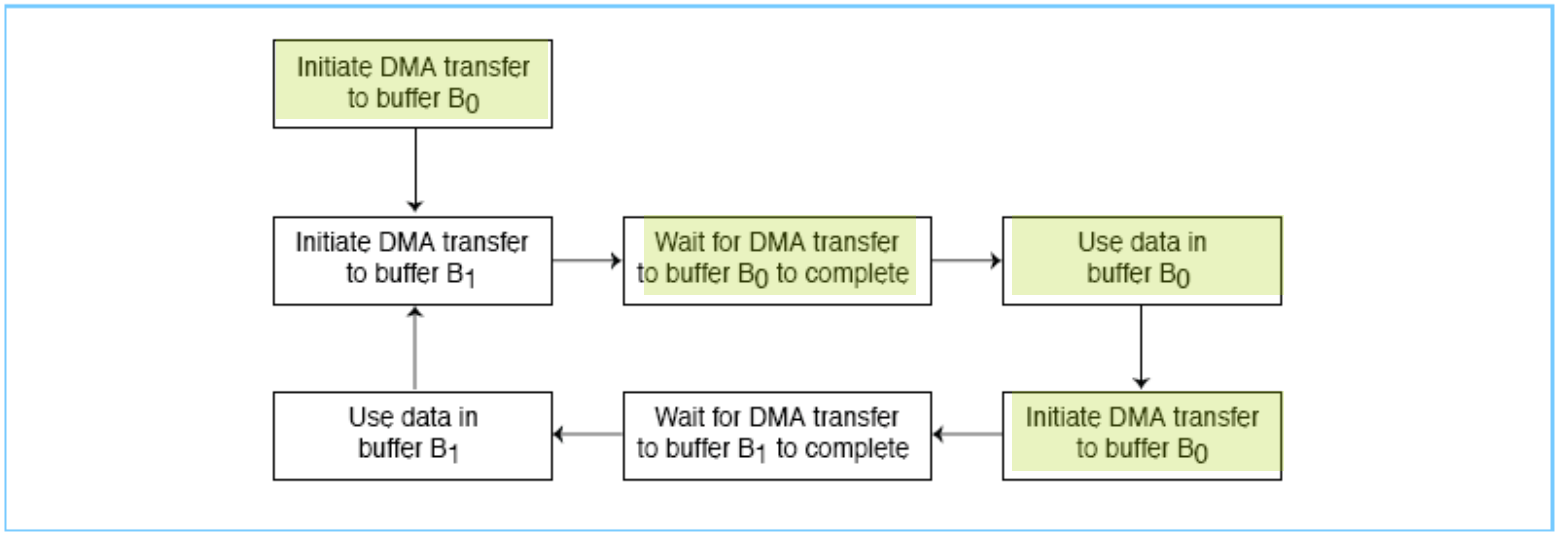



*Figure 3-7. DMA Transfers Using a Double-Buffering Method*



# Pseudo code

```
//initiate DMA transfer
DMA_Get()

While (--buffer) {

//initiate next DMA transfer until no data in buffer
DMA_Get()

//Wait for previous transfer to complete
DMA_barrier()

//use the data from the previous transfer
Use_data()
}

//wait for last transfer to complete
DMA_barrier()
//use the data from the last transfer
Use_data()
```