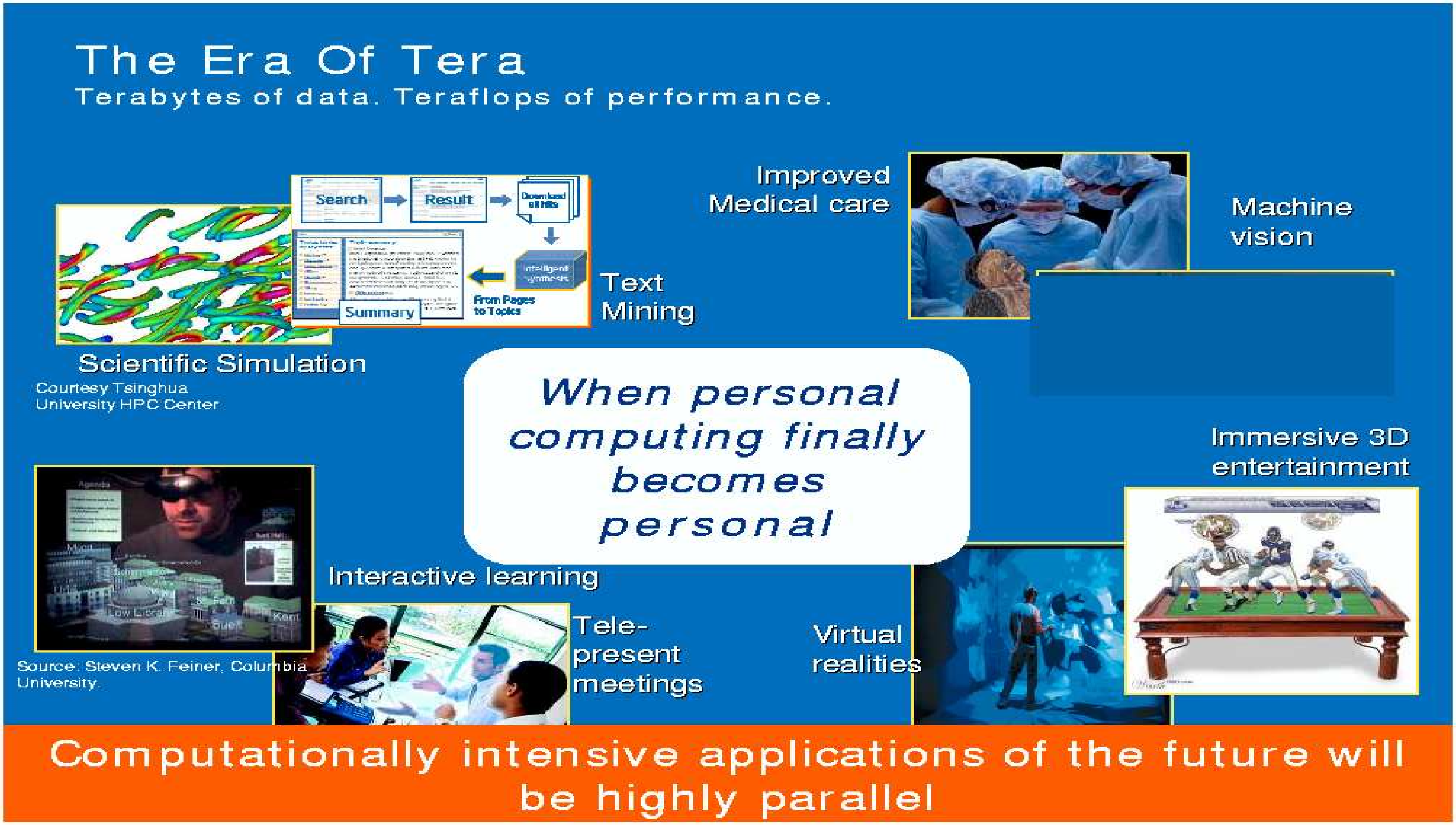
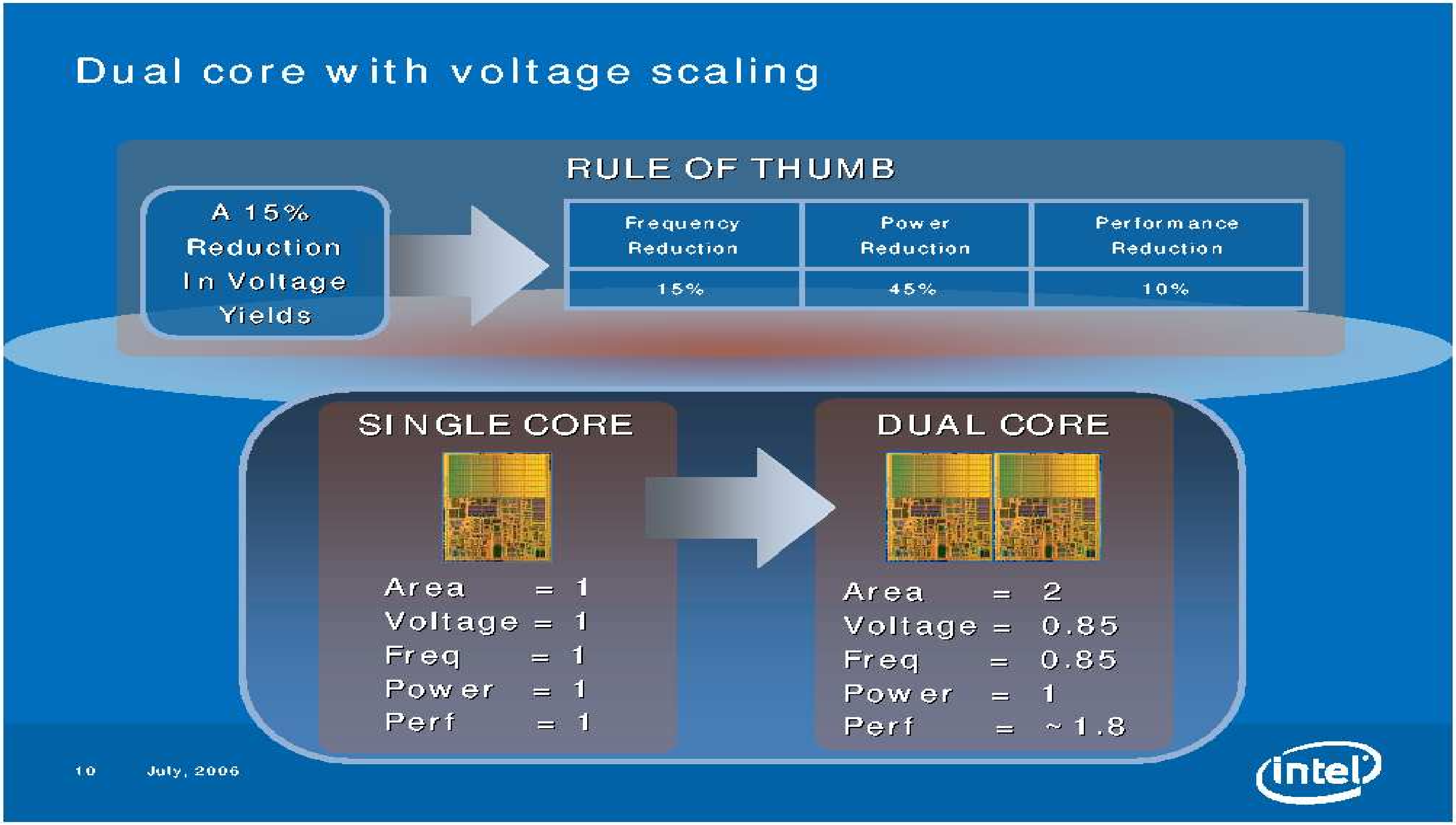
CS435/CS635
Parallel Scientific Computing: Models, Algorithms and Implementation

Marc Moreno Maza

February 19, 2008


The Era Of Tera
Terabytes of data. Teraflops of performance.
Improved Medical care
Machine vision
Text Mining
Scientific Simulation
Courtesy Tsinghua University HPC Center
When personal computing finally becomes personal
Immersive 3D entertainment
Interactive learning
Tele-present meetings
Virtual realities
Source: Steven K. Feiner, Columbia University.
Computationally intensive applications of the future will be highly parallel



Dual core with voltage scaling
RULE OF THUMB
A 15% Reduction In Voltage Yields
Frequency Reduction
Power Reduction
Performance Reduction
15%
45%
10%
SINGLE CORE
Area = 1
Voltage = 1
Freq = 1
Power = 1
Perf = 1
DUAL CORE
Area = 2
Voltage = 0.85
Freq = 0.85
Power = 1
Perf = ~1.8
10
July, 2006
intel

- **Technology is driving hardware venders to build multi-core processors.**
  - Single-processor machine cannot keep evolving following Moore's' law.
  - Moreover, some computations may require much more memory resources than we have on an usual PC.
  - Parallel applications will become mainstream
- **Applications will face (at least) three challenges:**
  - Scaling to $100,000$ processors and multi-core processors
  - Topology sensitive interconnection network
  - Memory wall

- Finding enough parallelism
  - Only part of an application seems parallel might be enough!
- Granularity
  - Applications need large enough work units to run fast in parallel
- Scalability
  - Performances should increase when more resources are available
- Locality and architecture-awareness
  - Memory traffic can limit the benefits of parallel execution
  - Parallel programs may run fast on some machines, slow on others!
- Load balance
  - unequal size tasks, insufficient parallelism make processors idle

$\Rightarrow$ Makes parallel programming harder than sequential programming.

- “Automatic” Parallelism in Modern Machines
  - Bit level parallelism: within floating point operations, etc.
  - Instruction level parallelism (ILP): multiple instructions execute per clock cycle
  - Memory system parallelism: overlap of memory operations with computation
  - OS parallelism: multiple jobs run in parallel on commodity SMPs
- **Limits to all of these** – for very high performance, the user needs to identify, schedule and coordinate parallel tasks.

- Given enough parallel work, the following issues are the biggest barrier to getting the desired speedup.
- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation
- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Trade-off: Algorithm needs
  - sufficiently large units of work to run fast in parallel (I.e. large granularity),
  - but not so large that there is not enough parallel work

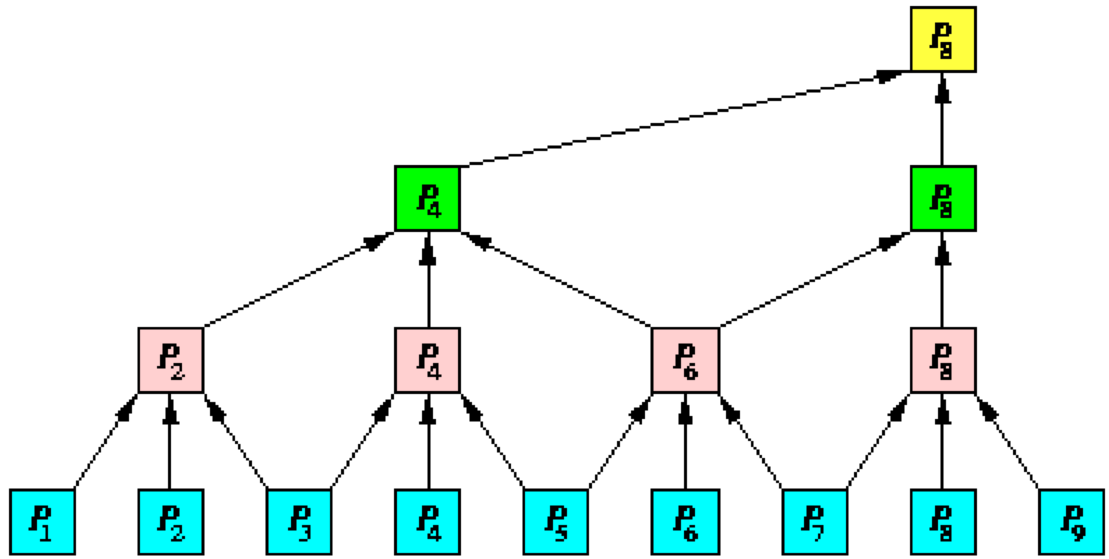
- Parallelizing algorithms: some examples
- A parallel program
- Memory traffic issues: a detailed example
- Topics covered in CS 855

Consider a tridiagonal linear system of order n:

$$
\begin{array}{cccccc}
\cdots & \cdots & \cdots & & & \cdots \\
a_{i-2}x_{i-2} & +\, b_{i-1}x_{i-1} & +\, c_i x_i & & = & e_{i-1} \\
& a_{i-1}x_{i-1} & +\, b_i x_i & +\, c_{i+1}x_{i+1} & = & e_i \\
& & a_i x_i & +\, b_{i+1}x_{i+1} \quad +\, c_{i+2}x_{i+2} & = & e_{i+1} \\
& & & \cdots \qquad \cdots \qquad \cdots & & \cdots
\end{array}
$$

For every even i replacing x_i with $-\frac{e_i - c_{i+1}x_{i+1} - a_{i-1}x_{i-1}}{b_i}$ leads to another tridiagonal system of order $n/2$:

$$
\begin{array}{cccccc}
\cdots & \cdots & \cdots & & & \cdots \\
A_{i-3}x_{i-3} & +\, B_{i-1}x_{i-1} & +\, C_{i+1}x_{i+1} & & = & E_{i-1} \\
& A_{i-1}x_{i-1} & +\, B_{i+1}x_{i+1} & +\, C_{i+3}x_{i+3} & = & E_{i+1} \\
& & \cdots \qquad \cdots & \cdots & & \cdots
\end{array}
$$



Observe that, on this example:

- the number of processors, here $p = n$, can be set such that
- the number of parallel steps, here $O(\log n)$, is known and small,
- processors activity (scheduling) is easy to organize,
- data-communication is not intensive.

Here's a second example of an algorithm that is *naturally parallel*. As for the *Karatsuba* algorithm, the trick relies on a smart use of distributivity:

(1) If $n = 1$ the matrices A and B are of the form (a) and (b) where $a, b \in \mathbf{Z}$, such that $A\,B$ equals $(a\,b)$.

(2) If $n > 1$ the matrices A and B can be decomposed as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \; B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

where A_{ij} and B_{ij} are square matrices of order $n/2$. Them, one computes the following 8 sums.

$$S_1 = A_{21} + A_{22} \qquad T_1 = B_{12} - B_{11}$$
$$S_2 = S_1 - A_{11} \qquad T_2 = B_{22} - T_1$$
$$S_3 = A_{11} - A_{21} \qquad T_3 = B_{22} - B_{12}$$
$$S_4 = A_{12} - S_2 \qquad T_4 = T_2 - B_{21}$$

(3) Next, one computes the following 7 matrix products by recursive calls to the algorithm.

$$P_1 = A_{11}\,B_{11} \qquad P_5 = S_1\,T_1$$
$$P_2 = A_{12}\,B_{21} \qquad P_6 = S_2\,T_2$$
$$P_3 = S_4\,B_{22} \qquad P_7 = S_3\,T_3$$
$$P_4 = A_{22}\,T_4$$

(4) Finally, one computes the following 7 sums:

$$U_1 = P_1 + P_2 \qquad U_5 = U_4 + P_3$$
$$U_2 = P_1 + P_6 \qquad U_6 = U_3 - P_4$$
$$U_3 = U_2 + P_7 \qquad U_7 = U_3 + P_5$$
$$U_4 = U_2 + P_5$$

and one returns:

$$\begin{pmatrix} U_1 & U_5 \\ U_6 & U_7 \end{pmatrix}.$$

Observe that, on this second example:

- one may need to duplicate data
- one has to be careful with data dependency
- how many processors can be busy at the same time?

$$\begin{cases} f_1 = x - 2 + (y-1)^2 \\ f_2 = (x-1)(y-1) + (x-2)y \\ f_3 = (x-1)z \end{cases}$$

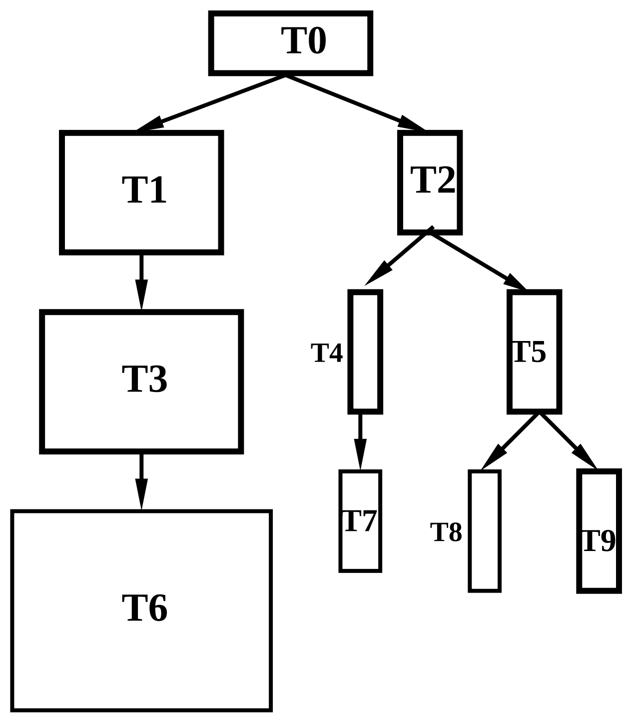
Factorizing f_3 leads to two sub-systems:

$$S_1 : \begin{cases} y = 0 \\ x = 1 \end{cases} \quad \text{and} \quad S_2 : \begin{cases} x - 1 + y^2 - 2y = 0 \\ (2y-1)x + 1 - 3y = 0 \\ z = 0 \end{cases}$$

The sub-system S_1 is solved. Continuing with S_2 leads finally to

$$\begin{cases} z = 0 \\ y = 0 \\ x = 1 \end{cases}, \begin{cases} z = 0 \\ y = 1 \\ x = 2 \end{cases} \quad \text{and} \quad \begin{cases} z = 0 \\ 2y = 3 \\ 4x = 7 \end{cases}$$

- Very irregular tasks (CPU time, memory, data-communication)
- For most systems with rational number coefficients, in theory and in practice, the solution set consists of one main piece and possibly a few tiny pieces.



```
fib(n)
       if n < 2
       then return n
       x <- spawn fib(n-1)
       y <- spawn fib(n-2)
       sync
       return (x+y)
```

- A `spawn` instruction allows the next instruction of the parent program to be executed at the same time by a child, which is equivalent to `fork` in UNIX programming.
- A `sync` instruction requires that all locally spawned children finish their work before proceeding with the computation (equivalent to UNIX `join`).

Our syntax describes **logical parallelism** only. We do not state which tasks are assigned to which processors, leaving it to **scheduler**.

We model the instruction stream of a program as a **computation directed acyclic graph (DAG)**. Vertices are **threads**, which is a maximal sequence of instructions not containing any parallel control. We have three threads here:

Thread A: the `if` statement up to the first `spawn`.

Thread B: the computation of `n-2` up to the second `spawn`.

Thread C: the `x + y` operation before `return`.

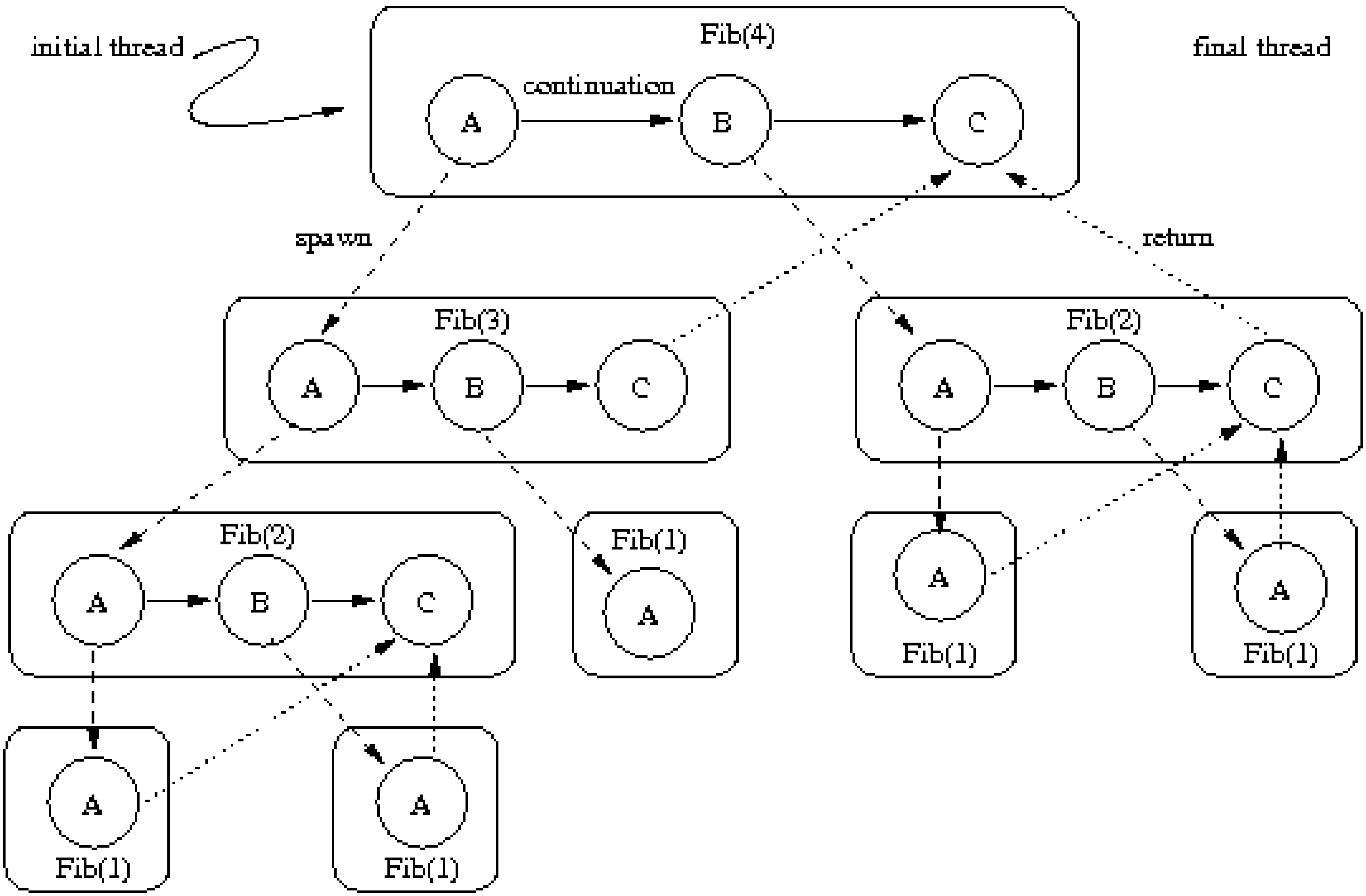
We define several performance measures to evaluate various scheduler algorithms. We assume an ideal situation: no cache issues, no inter-processor costs:

T_p is the minimum running time on p processors.

T_1 is called the **work**.

T_∞ is the minimum running time on an infinite number of processors.

Instructions Streams as a DAG



On the previous figure:

- Each solid arrow represents a `continuation`.
- Each dashed arrow represents a `spawn`.
- Each dotted arrow represents a `return`.

Observe that we have $T_1 = 17$ and $T_\infty = 8$.

Proposition 1 *Assuming that all threads run in unit time, the longest path in the DAG is equal to T_∞. For this reason, T_∞ is referred to as the critical path length. When threads do not run in unit time, then the vertices of the DAG need to be weighted by their individual runtimes.*

Proposition 2 *We have: $T_p \geq T_1/p$.*

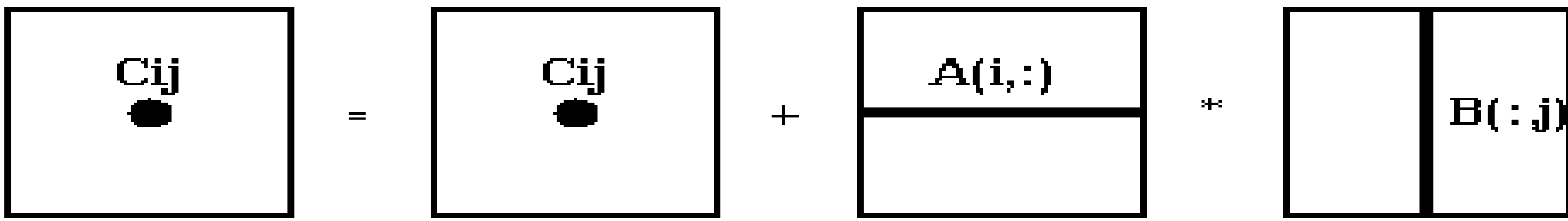
Indeed, in the best case, p processors can do p works per unit of time.

Proposition 3 *We have: $T_p \geq T_\infty$.*

Indeed, $T_p < T_\infty$ contradicts the definitions of T_p and T_∞.

- We aim at multiplying large square matrices of order n. We assume that we are using floating point numbers.
- Assume we have two levels of memory: slow and fast.
- The input and output matrices reside entirely in slow memory.
- The small and fast memory has size M words with $M << n^2$ but $M \geq 4n$.
- Each word is read from slow memory individually and we have complete control over which words are transferred between the two levels.
- We denote by m the number of read/write access to slow memory
- Let $f = 2n^3$ the total number of operations on floating point numbers.
- Finally, assume that one read/write access to slow memory is much more expensive than one operation on floating point numbers.

Unblocked matrix multiplication




```
for i=1 to n
   {Read row i of A into fast memory}
   for j=1 to n
      {Read C(i,j) into fast memory}
      {Read column j of B into fast memory}
      for k=1 to n
          C(i,j)=C(i,j) + A(i,k)*B(k,j)
      end for
      {Write C(i,j) back to slow memory}
   end for
end for
```

```
for i=1 to n
   {Read row i of A into fast memory}
   for j=1 to n
      {Read C(i,j) into fast memory}
      {Read column j of B into fast memory}
      for k=1 to n
          C(i,j)=C(i,j) + A(i,k)*B(k,j)
      end for
      {Write C(i,j) back to slow memory}
   end for
end for
```

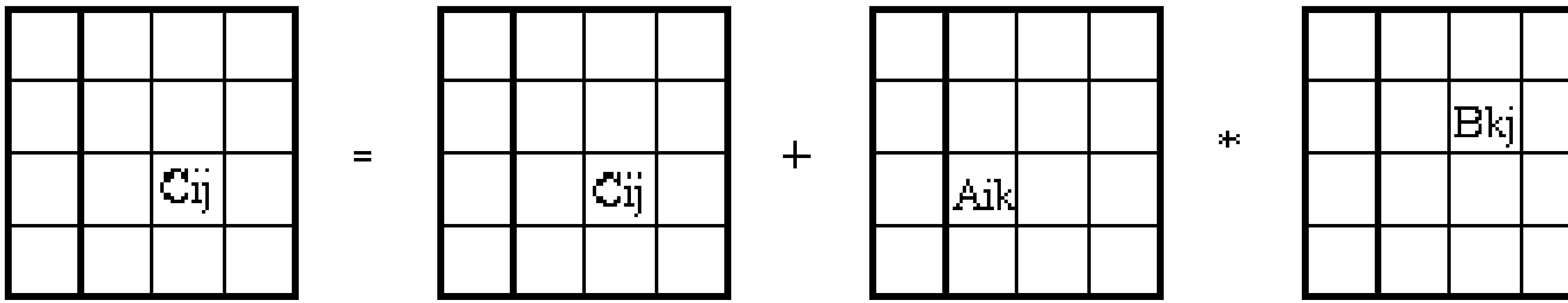
Let's count the number of accesses to the slow memory. For A, B and C respectively: $n^2, n^3, 2n^2$. Leading to:

$$m = n^3 + n^2 + 2n^2 = n^3 + 3n^2$$

and

$$q = \frac{f}{m} = \frac{2n^3}{n^3 + 3n^2} \sim 2.$$

Blocked matrix multiplication



$$\boxed{C_{ij}} = \boxed{C_{ij}} + \sum_{k=1}^{n} \boxed{A_{ik}} * \boxed{B_{kj}}$$

```
for i=1 to N
    for j=1 to N
        {Read Cij into fast memory}
        for k=1 to N
            {Read Aik into fast memory}
            {Read Bkj into fast memory}
            Cij = Cij + Aik * Bkj
        end for
        {Write Cij back to slow memory}
    end for
end for
```

```
for i=1 to N
    for j=1 to N
        {Read Cij into fast memory}
        for k=1 to N
            {Read Aik into fast memory}
            {Read Bkj into fast memory}
            Cij = Cij + Aik * Bkj
        end for
        {Write Cij back to slow memory}
    end for
end for
```

Each matrix is regarded as N-by-N blocks of size $n/N \times n/N$.

For A, B and C we have respectively: $Nn^2, Nn^2, 2n^2$ accesses.

$$m = Nn^2 + Nn^2 + 2n^2 = (2N+2)n^2.$$

and

$$q = \frac{f}{m} = \frac{2n^3}{(2N+2)n^2} \sim \frac{n}{N} \sim \sqrt{\frac{M}{3}}.$$

Since 3 blocks must be in fast memory, so $N \sim n\sqrt{\frac{3}{M}}$.

- Performance and scalability of parallel algorithms.
- Parallel machine models.
- Parallel complexity theory.
- Scheduling and load balancing.
- Routing and broadcast algorithms.
- Solving systems of linear equations in parallel.
- Parallel programming environments.
- Code optimization for parallelism and locality.
- Project presentations.

- Undergraduate student regime:
  - 2 assignments (40 %)
  - 3 Quizzes (20 %)
  - a research project (40 %) based on one article and leading to a
    - a **short** report and
    - a presentation in class.
- Graduate student regime: undergraduate student regime or
  - 3 Quizzes (20 %)
  - a research project (80 %) based on **several** articles and leading to a
    - a **detailed** report and
    - a presentation in class.

References and Acknowledgments

Most of the material of these course notes are inspired by:

- *Parallel Algorithms* (WISM 459, 2005/2006) by Rob Bisseling.
  `http://www.math.uu.nl/people/bisselin/pas1.html`
- *Topics in Parallel Computing* (CS 838, Univ. of Wisconsin-Madison, Spring 1999) by Pavel Tvrdik.
  `http://www.cs.wisc.edu/ tvrdik/cs838.html`
- *Parallel Algorithms* (CS 662, San Diego Univ., Spring 1996) by Roger Whitney.
  `http://www.eli.sdsu.edu/courses/spring96/cs662/notes/`
- *Applications of Parallel Computers* by James Demmel.
  `http://www.cs.berkeley.edu/ demmel/cs267_Spr06/`
- *Parallel Systems* by Charles Leiserson
  `http://theory.csail.mit.edu/classes/6.895/fall03/`
- Geoff Lowney (Intel Fellow, Microprocessor Architecte)