Parallel Programming Principle and Practice

Lecture 7 —Performance optimization on single processor computer

Outline

- Why performance optimization on single machine
- Cache effect
  - computational intensity
  - locality and blocked/tiled algorithm

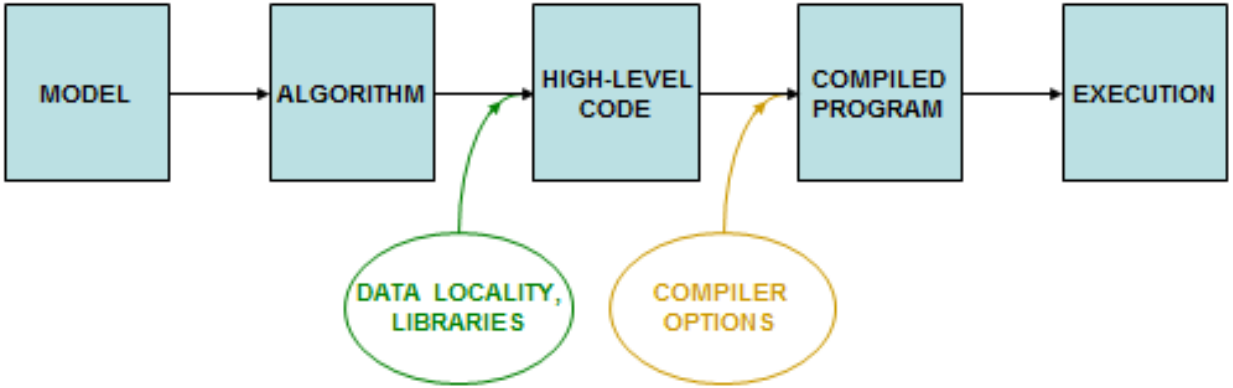
Performance optimization on single processor computer

why performance optimization on single machine

Why performance optimization on single processor

- Attaining good global performance
  - First, writing a parallel program and making sure it is arranged correctly for the target parallel architecture
  - Then, depends on making sure that one's single-CPU implementation in the program translates well into single-CPU machine code
    - how the algorithm is realized
    - how it is expressed in high-level language
    - how the compiler converts the high-level program into a specific sequence of operations on data

Optimization on single processor



$$T(n, p) \geq \sigma(n) + \varphi(n)/p + \kappa(n, p)$$

- The speed at which you can compute is bounded by:

  (the clock rate of the cores) x **(the amount of parallelism you can exploit)**

- Therefore, optimizing the performance of a single machine is helpful to improve the computing speed

Optimization on single processor

- The ability to achieve good single-core performance can be broken down into three basic principles
  - Creating a layout of data structures that results in **efficient memory accesses**
  - Employing optimized HPC libraries where possible in your program
  - Using appropriate compiler options when building your application

performance optimization on single processor computer

cache effect(快速缓存效应)

Computational intensity

- The **computational intensity** of an algorithm is the average number of flops per slow memory access

$$q = f / m$$

- f = number of arithmetic operations (e.g. floating-point adds and multiplies)
- m = number of memory elements (words) moved between fast and slow memory

***tm* = time per slow memory operation**

tf* = time per arithmetic operation << *tm

Computational intensity

- Assume just 2 levels in the memory hierarchy, fast and slow
- All data initially in slow memory
- Minimum possible time = $f * tf$ when all data in fast memory
- Actual time = $f * tf + m * tm = f * tf * (1 + tm/tf * 1/q)$

- Larger q means time closer to minimum $f * tf$
- tm/tf – machine balance (key to machine efficiency)
- **$q \geq tm/tf$ needed to get at least half of peak speed**

Locality (局部性)

- A type of predictable behavior that occurs in computer systems
  - A tendency of a processor to access the same set of memory locations repetitively over a short period of time
  - Two basic types of reference locality
    - **Temporal locality** (时间局部性)
      - the reuse of specific data and/or resources within a relatively small time duration
    - **Spatial locality** (空间局部性)
      - all those instructions which are stored nearby to the recently executed instruction have high chances of execution
      - It refers to the use of data elements(instructions) which are relatively close in storage locations

Temporal Locality Example

Temporal locality

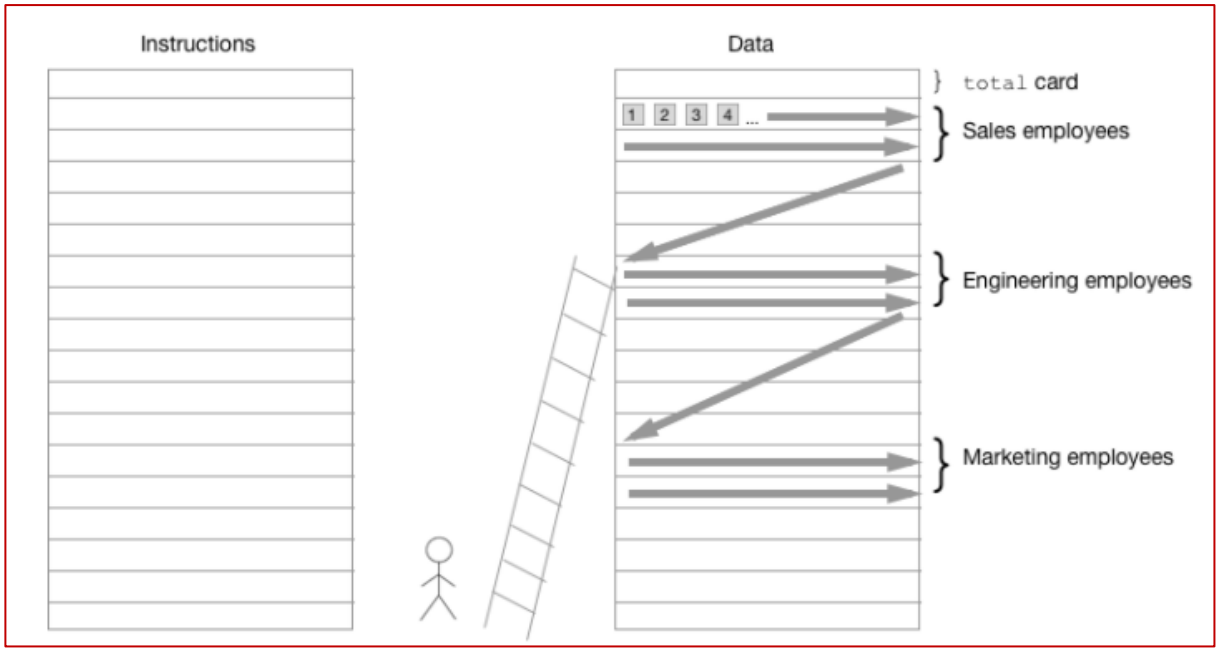
- accessing the same piece of data frequently over a short period of time
- "Temporal" means "time" and "locality" means "close to"

every line represents the filing clerk accessing a particular piece of data (eg. total)

On the left, there is temporal locality. On the right, there isn't.

Spatial Locality Example

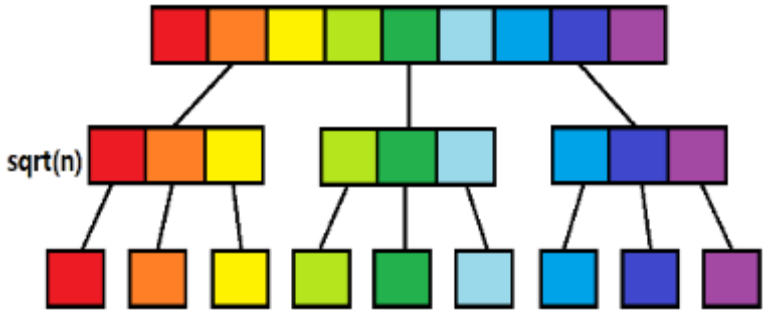
- **spatial locality**
  - it involves accessing cards that are close to one another as opposed to jumping around from one area to the next



Suppose the instructions have the filing clerk accessing employee cards from left to right, top to bottom

blocked/tiled algorithm (分块算法)

- 序列长度为*n*序列，分成*x*叉树形式（每层有*x*个块），**可以分成 $log_x(n)$ 层**
  - 如果采用*sqrt(n)*叉树，就有*2*层(分块)
  - 如果采用*2*叉树，就有$log_2 n$层（线段树）



- 时间复杂度
  - *n*序列上，修改是*O(n)*的复杂度，查询是*O(n)*，算法的复杂度*O(m*n)*
  - 线段树上做，修改与查询都为 $O(\log_x(n))$，算法的复杂度$O(m*\log_x(n))$
  - 根号算法：修改和查询复杂度为 *O(sqrt(n))*，算法的复杂度*O(m*sqrt(n))*

Parallel Programming Principle and Practice

Lecture 8 — Two-stage design process: stage 1

Outline

- Stage 1: machine independent (与具体机器无关)
  - analysis of parallelism (问题并行性分析)
    - ✓ Partitioning (任务分割)
      - associative law (叠加性)
      - data parallelism (数据并行)
      - data decomposition (数据分解)
      - task decomposition (任务分解)
    - ✓ data dependency
    - ✓ fine grained parallelism
    - ✓ parallel complexity
    - ✓ scalability

$$T(n, p) \geq \sigma(n) + \varphi(n)/p + \kappa(n, p)$$

Two-stage design process

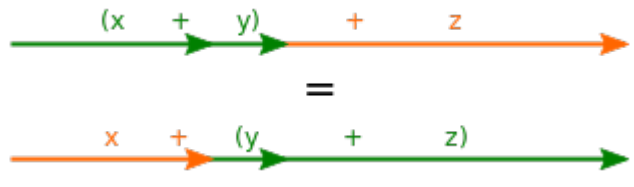
Stage 1: machine independent

Associative law (叠加性)

- In mathematics, the **associative law** is a property of some binary operations, which means that rearranging the parentheses in an expression will not change the result
- In propositional logic, **associativity** is a valid rule of replacement for expressions in logical proofs
- In computer, it means that a task can be divided into different parts

$$(2 + 3) + 4 = 2 + (3 + 4) = 9$$

$$2 \times (3 \times 4) = (2 \times 3) \times 4 = 24.$$



Examples

- The concatenation of the three strings “hello”, “ ”, “world” can be computed by
  - (“hello”+“ ”)+“world” or “hello”+(“ ”+“world”)
  - The two methods produce the same result; string concatenatin is associative
- In computer science, the addition and multiplication of floating point numbers is not associative, as rounding errors are introduced when dissimilar-sized values are joined together.
  - To illustrate this, consider a floating point representation with a 4-bit mantissa:

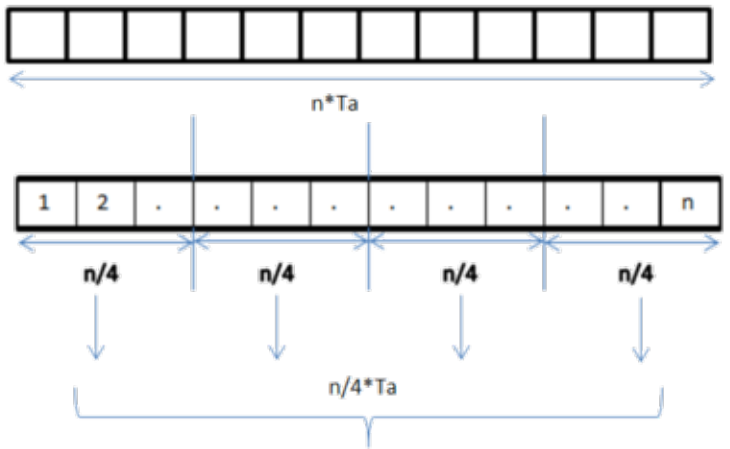
$$(1.000_2\times2^0 + 1.000_2\times2^0) + 1.000_2\times2^4 = 1.000_2\times2^1 + 1.000_2\times2^4 = 1.001_2\times2^4$$
$$1.000_2\times2^0 + (1.000_2\times2^0 + 1.000_2\times2^4) = 1.000_2\times2^0 + 1.000_2\times2^4 = 1.000_2\times2^4$$

Data parallelism (数据并行)

- Data parallelism is parallelization across multiple processors in parallel computing environments
  - It focuses on distributing the data across different nodes, which operate on the data in parallel
  - It can be applied on regular data structures like **arrays** and **matrices** by working on each element in parallel
  - It contrasts to **task parallelism** as another form of parallelism

Data parallelism (数据并行)

- A data parallel job on an array of n elements can be divided equally among all the processors
  - sum all the elements of the given array and the time for a single addition operation is *Ta* time units.
  - sequential execution : $n \times Ta$
  - a data parallel job on 4 processors: $(n/4) \times Ta +$ merging overhead time units
  - Speedup: 4 over sequential execution



Example: matrices multipication

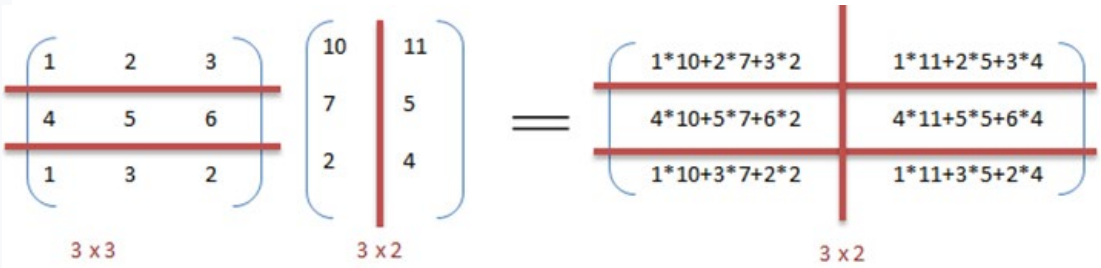
- multiplication calculates the dot product of two matrices A, B and stores the result into the output matrix C

```
// Matrix multiplication
for (i = 0; i < row_length_A; i++)
{
    for (k = 0; k < column_length_B; k++)
    {
        sum = 0;
        for (j = 0; j < column_length_A; j++)
        {
            sum += A[i][j] * B[j][k];
        }
        C[i][k] = sum;
    }
}
```

Example: data parallelism

- divide matrix A and B into blocks along rows and columns respectively
- calculate every element in matrix C **individually** thereby making the task parallel
- A[m × n] dot B [n × k] can be finished in $O(n)$ instead of $O(m * n * k)$ when executed in parallel using m*k processors

```
// Matrix multiplication in parallel
#pragma omp parallel for schedule(dynamic,1) collapse(2)
for (i = 0; i < row_length_A; i++){
    for (k = 0; k < column_length_B; k++){
        sum = 0;
        for (j = 0; j < column_length_A; j++){
            sum += A[i][j] * B[j][k];
        }
        C[i][k] = sum;
    }
}
```

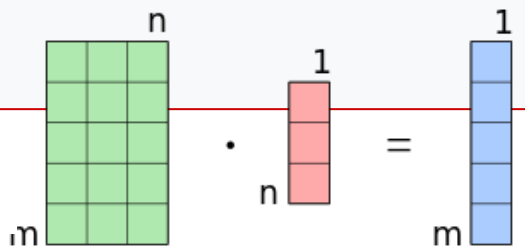


Example: data parallelism

- matrix-vector multiplication algorithm in OpenCL C

```
// Multiplies A*x, leaving the result in y.
// A is a row-major matrix, meaning the (i,j) element 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 id, used as the row index
    __global float const *a = &A[i*ncols];    // Pointer to the i'th row
    float sum = 0.f;                          // Accumulator for dot product
    for (size_t j = 0; j < ncols; j++) {
        sum += a[j] * x[j];
    }
    y[i] = sum;
}
```

$$y_i = a_{i,:} \cdot x = \sum_j a_{i,j} x_j.$$



Decomposition (分解)

- Decomposition in computer science, also known as factoring, is breaking a complex problem or system into parts that are easier to conceive, understand, program, and maintain
- There are different types of decomposition defined in computer sciences
  - Data decomposition
  - Task decomposition

Data decomposition example

- As a simple example of data decomposition, consider the addition of two vectors, A[1..N] and B[1..N], to produce the result vector, C[1..N]
  - P processes: **data partitioning** involves the allocation of N/P elements of each vector to each process, which computes the corresponding N/P elements of the resulting vector
  - Done ``statically ''
    - each process knows a priori (at least in terms of the variables N and P) its share of the workload
  - Done ``dynamically''
    - a control process (e.g., the master process) allocates subunits of the workload to processes as and when they become free

Data decomposition

- Issue: input and output
  - how do the processes described above receive their workloads, and what do they do with the result vectors?
    - Individual processes generate their own data internally, for example, using random numbers or statically known values. This is possible only in very special situations or for program testing purposes
    - Individual processes independently input their data subsets from external devices. This method is meaningful in many cases, but possible only when parallel I/O facilities are supported
    - **A controlling process sends individual data subsets to each process**. This is the most common scenario, especially when parallel I/O facilities do not exist

Task decomposition

Definition

- Task Decomposition is the division of a larger (root) task into smaller, more manageable elements or sub-tasks to deal with the root task at the lowest possible level and therefore with higher simplicity
- Task decomposition can be presented as an attempt to make broader tasks simpler and more understandable through creating hierarchies of interdependent sub-tasks
- For example: **parallelism in distributed-memory environments may be achieved by partitioning the overall workload in terms of different operations**

Task decomposition

- There are two general steps in decomposing tasks
  - Modelling
    - A conventional task model highlights a hierarchical structure of the root task and outlines relationships between this task and its sub-tasks
    - It describes one or more possible scenarios for the task's performance
  - Charting
    - Creating task flowcharts helps get a graphical design of the root task and its sub-tasks
    - Task flowchart is based on related task models and graphically displays task breakdown

Example

- the three stages of typical program execution: input, processing, and result output
  - In function decomposition, such an application may consist of three separate and distinct programs, each one dedicated to one of the three phases
  - Parallelism is obtained by concurrently executing the three programs and by establishing a "pipeline" (continuous or quantized) between them
  - In such a scenario, data parallelism may also exist within each phase

data dependency (数据依赖)

- A data dependency is a situation in which a program statement (instruction) refers to the data of a preceding statement
- The technique used to discover data dependencies among statements (or instructions) is called **dependence analysis**

- There are three types of dependencies: data, name, and control
  - Flow dependency (True dependency)
  - Anti-dependency
  - Output dependency

flow dependency (True dependency)

- A **Flow dependency**, also known as a data dependency or true dependency or **read-after-write (RAW),** occurs when an instruction depends on the result of a previous instruction

```
1. A = 3
2. B = A
3. C = B
```

- Instruction 3 is truly dependent on instruction 2
- Instruction 2 is truly dependent on instruction 1
- Instruction level parallelism is therefore not an option in this example

anti-dependency

- An **anti-dependency**, also known as **write-after-read (WAR),** occurs when an instruction requires a value that is later updated

```
1.  B = 3
2.  A = B + 1
3.  B = 7
```

- instruction 2 anti-depends on instruction 3
- the ordering of these instructions cannot be changed, nor can they be executed in parallel (possibly changing the instruction ordering), as this would affect the final value of A

- Example

```
MUL R3,R1,R2
ADD R2,R5,R6
```

- It is clear that there is anti-dependence between these 2 instructions

anti-dependency

- Renaming of variables could remove the dependency

```
1.  B = 3
2.  A = B + 1
3.  B = 7
```



```
1.  B = 3
N.  B2 = B
2.  A = B2 + 1
3.  B = 7
```

- The anti-dependency between 2 and 3 has been removed, meaning that these instructions may now be executed in parallel.
- However, the modification has introduced a new dependency
  - instruction 2 is now truly dependent on instruction N, which is truly dependent upon instruction 1
  - As flow dependencies, these new dependencies are impossible to safely remove

output dependency

- An **output dependency**, also known as **write-after-write (WAW)**, occurs when the ordering of instructions will affect the final output value of a variable.

```
1. B = 3
2. A = B + 1
3. B = 7
```

- there is an output dependency between instructions 3 and 1
- changing the ordering of instructions in this example will change the final value of A
- these instructions cannot be executed in parallel

input dependency

- An **input dependency**, also known as **Read-After-Read (RAR)**

```
S1      y  := x + 3
S2      z  := x + 5
```

- A statement S2 is input dependent on S1 if and only if S1 and S2 read the same resource and S1 precedes S2 in execution
- S2 and S1 both access the variable x
- This dependence does not prohibit reordering

Loop With Dependence

- any two successive iterations, *i* and *i+1*, will write and read the same variable *sum*
- in order for these two iterations to execute in parallel some form of locking on the variable would be required

```
for (i=1; i < 1000; i++) {
    sum = sum + a[i]; /* S1 */
}
```

- the use of locks imposes overhead that might slowdown the program
- The C compiler will not ordinarily parallelize the loop in the example above because there is a data dependence between two iterations of the loop

Loop Without Dependence

- Consider another example

```
for (i=1; i < 1000; i++) {
    a[i] = 2 * a[i]; /* S1 */
}
```

- each iteration of the loop references a different array element
- different iterations of the loop can be executed in any order
- They may be executed in parallel without any locks because no two data elements of different iterations can possibly interfere

Example：Compiler

- The analysis performed by the compiler to determine whether two different iterations of a loop could reference the same variable is called **data dependence analysis**
  - three outcomes
    - ✓ There is a dependence, in which case, executing the loop in parallel is not safe
    - ✓ There is no dependence, in which case the loop may safely execute in parallel using an arbitrary number of processors
    - ✓ The dependence cannot be determined. The compiler assumes, for safety, that a dependence might prevent parallel execution of the loop and will not parallelize the loop

Example：Compiler

Example： Loop That Might Contain Dependencies

```
for (i=1; i < 1000; i++) {
    a[b[i]] = 2 * a[i];
}
```

- whether two iterations of the loop write to the same element of array a depends on whether array b contains duplicate elements
- Unless the compiler can determine this fact, it must assume there might be a dependence and not parallelize the loop

fine grained parallelism (细粒度并行)

- In fine-grained parallelism, a program is broken down to a large number of small tasks.
  - These tasks are assigned individually to many processors
  - The amount of work associated with a parallel task is low and the work is evenly distributed among the processors
  - fine-grained parallelism facilitates load balancing
- As each task processes less data, the number of processors required to perform the complete processing is high
  - This in turn, increases the communication and synchronization overhead
- Fine-grained parallelism is best exploited in architectures which support fast communication
  - Shared memory architecture which has a low communication overhead is most suitable for fine-grained parallelism.
  - It is difficult for programmers to detect parallelism in a program, therefore, it is usually the compilers' responsibility to detect fine-grained parallelism

fine grained parallelism

- Granularity is closely tied to the level of processing. fine grained parallelism can be broken down into 2 levels
  - Instruction level
  - Loop level

- **For example**
  - Typical grain size at instruction-level is 20 instructions
  - while the grain-size at loop-level is 500 instructions

Levels	Grain Size	Parallelism
Instruction level	Fine	Highest
Loop level	Fine	Moderate

parallel complexity (并行复杂度)

- the analysis of parallel algorithms is the process of finding the **computational complexity** of algorithms executed in parallel
  - the amount of **time**, **storage**, or other resources needed to execute them
  - the analysis of the behavior of multiple cooperating threads of execution
  - One of the primary goals of parallel analysis is to understand how a parallel algorithm's use of resources (speed, space, etc.) changes as the number of processors is changed

parallel complexity(并行复杂度)

- Suppose computations are executed on a machine that has p processors. Let T_p denote the time that expires between the start of the computation and its end
  - The *work* of a computation executed by p processors is the total number of primitive operations that the processors perform
    - ✓ Ignoring communication overhead from synchronizing the processors, this is equal to the time used to run the computation on a single processor, denoted T_1
  - The *cost* of the computation is the quantity pT_p
    - ✓ This expresses the total time spent, by all processors, in both computing and waiting

parallel complexity (并行复杂度)

- The *depth* or *span* is the length of the longest series of operations that have to be performed sequentially due to data dependencies (the *critical path*)
  - The *depth* may also be called the critical path length of the computation
  - Minimizing the *depth/span* is important in designing parallel algorithms, because the *depth/span* determines the shortest possible execution time
  - Alternatively, the span can be defined as the time T_∞ spent computing using an idealized machine with an infinite number of processors

parallel complexity(并行复杂度)

- Several useful results follow from the definitions of work, span and cost
  - *Work law.* The cost is always at least the work: $pT_p \geq T_1$. This follows from the fact that p processors can perform at most p operations in parallel
  - *Span law.* A finite number p of processors cannot outperform an infinite number, so that $T_p \geq T_\infty$

parallel complexity(并行复杂度)

- Using these definitions and laws, the following measures of performance can be given
  - *Speedup* is the gain in speed made by parallel execution compared to sequential execution: $S_p = T_1 / T_p$.
    - ✓ The situation T_1 / T_p= p is called perfect linear speedup.
  - *Efficiency* is the speedup per processor, S_p / p
  - *Parallelism* is the ratio T_1/T_∞
    - ✓ It represents the maximum possible speedup on any number of processors
    - ✓ By the span law, the parallelism bounds the speedup: if p > T_1/T_∞, then:

$$\frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} < p$$

Scalability(可扩展性)

- scalability has two usages
  - The ability of a computer application or product (hardware or software) to continue to function well when it (or its context) is changed in size or volume in order to meet a user need
    - ✓ Typically, the rescaling is to a larger size or volume
    - ✓ The rescaling can be of the product itself (for example, a line of computer systems of different sizes in terms of storage, RAM, and so forth) or in the scalable object's movement to a new context (for example, a new operating system)
  - It is the ability not only to function well in the rescaled situation, but to actually take full advantage of it.
    - For example, an application program would be scalable if it could be moved from a smaller to a larger operating system and take full advantage of the larger operating system in terms of performance (user response time and so forth) and the larger number of users that could be handled

Scalability (可扩展性)

- Example
  - John Young in his book *Exploring IBM's New-Age Mainframes* describes the **RS/6000 SP system** as one that delivers scalability ("the ability to retain performance levels when adding additional processors")

IBM RS/6000 SP

- Another example: In printing, scalable fonts are fonts that can be resized smaller or larger using software without losing quality

Example

- Scalability is the ability of a program to scale.
  - For example, if you can do something on a small database (say less than 1000 records), a program that is highly scalable would work well on a small set as well as working well on a large set (say millions, or billions of records).

```
/**
    read all the data from
    small database with 1000 records
**/
for (int i=0; i<1000; i++){
    data = read(database);
    save(data);

    do( ...something else ...)

}
```

bad scalability: it does not work well when it is moved into other database without 1000 records.

```
/**
    read all the data from
    dataset with any number of records
**/
//get the number of record
nums  = getNum(database);
for (int i=0; i<nums; i++){
    data = read(database);
    save(data);

    do( ...something else ...)

}
```

good scalability: it work well when it is moved into any database.

Conclusion

- Why performance optimization on single machine
- Cache effect
  - computational intensity
  - locality and blocked/tiled algorithm
- Stage 1: machine independent
  - analysis of parallelism (问题并行性分析)
    - ✓ Partitioning (任务分割)
      - associative law (叠加性)
      - data parallelism (数据并行)
      - data decomposition (数据分解)
      - task decomposition (任务分解)
    - ✓ data dependency
    - ✓ fine grained parallelism
    - ✓ parallel complexity
    - ✓ scalability