Анализ и оптимизация на софтуерни приложения

Александър Пенев

Васил Василев

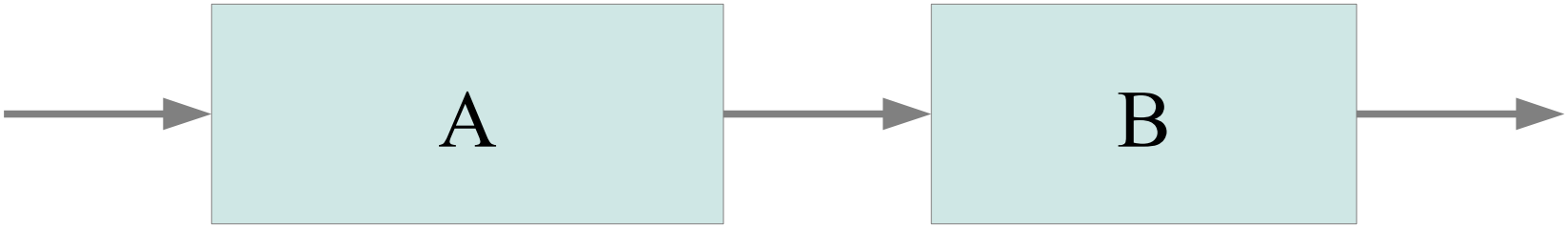
Паралелизация

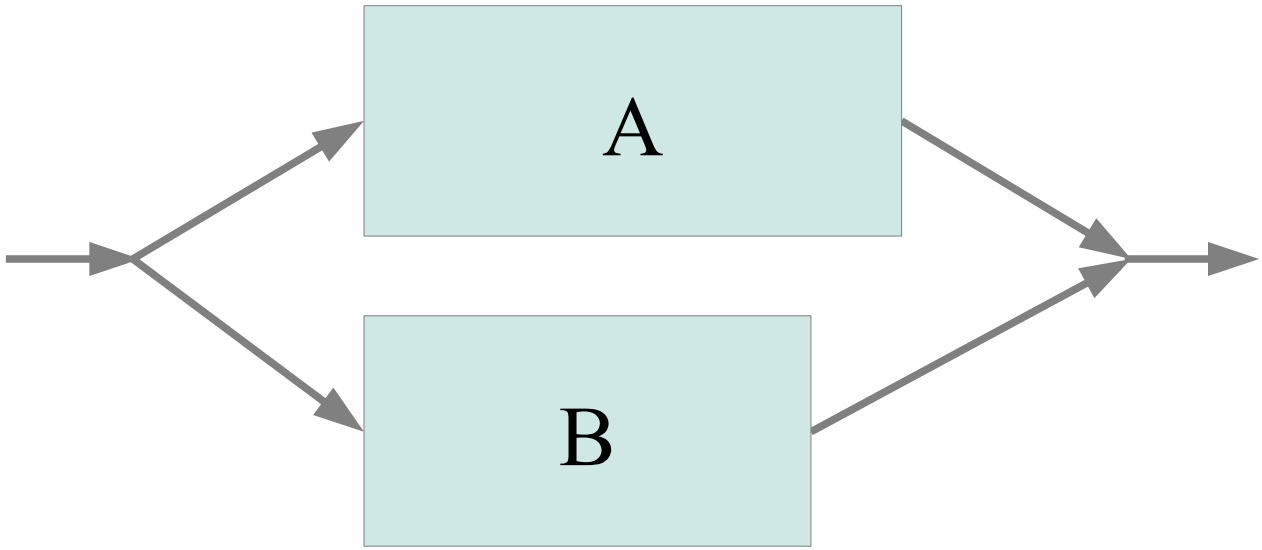
Съдържание

Какво е паралелизация?

- Паралелизация (parallelization) е процес на преобразуване на програма изпълняваща стъпките на алгоритмите си последователно в програма изпълняваща ги (там където е възможно) паралелно/едновременно
- Паралелното изпълнение може да е с използването на SIMD инструкции (векторизация), много нишки, много процеси, много компютри
- Обикновено се използва повече от едно ядро/процесор

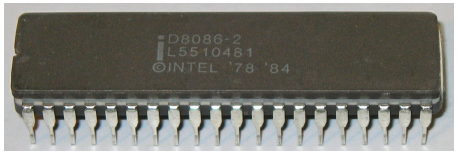
Какво е паралелизация?





Защо?

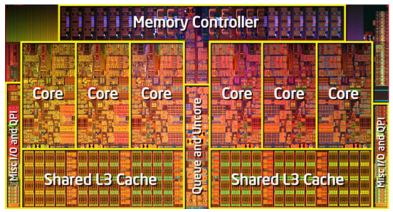
Преди (70-те години на миналия век)



Intel 8086

Сега (2013 г.)

nVidia Tesla / GPGPU



Intel i7 Процесор

Datacenter

Как се използва

- PThreads
- Threading Building Blocks (TBB)
- Cilk++
- OpenMP
- OpenCL, CUDA
- ...

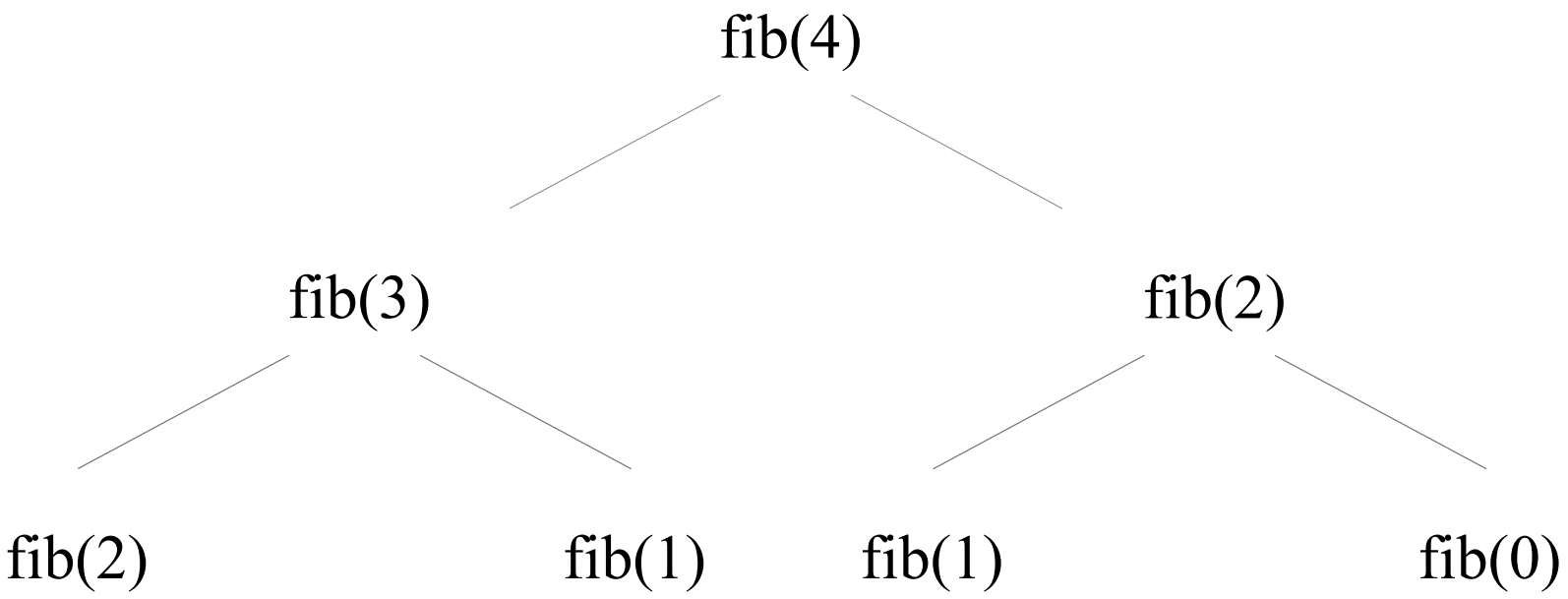
Пример 1

```
int fib(int n) {
  if (n < 2) return n;

  int n1, n2;

  #pragma omp task shared(n1)
  n1 = fib(n - 1);
  #pragma omp task shared(n2)
  n2 = fib(n - 2);

  #pragma omp taskwait

  return n1 + n2;
}
```

Указание за компилатора

Ако компилатора не поддържа автоматична паралелизация (OpenMP), то програмата се компилира както до сега

Пример – анализ



Важни характеристики

- T_P – Времето за изпълнение на P процесора

  Времето за изпълнение на алгоритъма на повече от един процесор може да съвпада с това за изпълнението на повече процесори, ако алгоритъма не е паралелизиран

- T_∞ – Време за изпълнение на т.н. критичен път

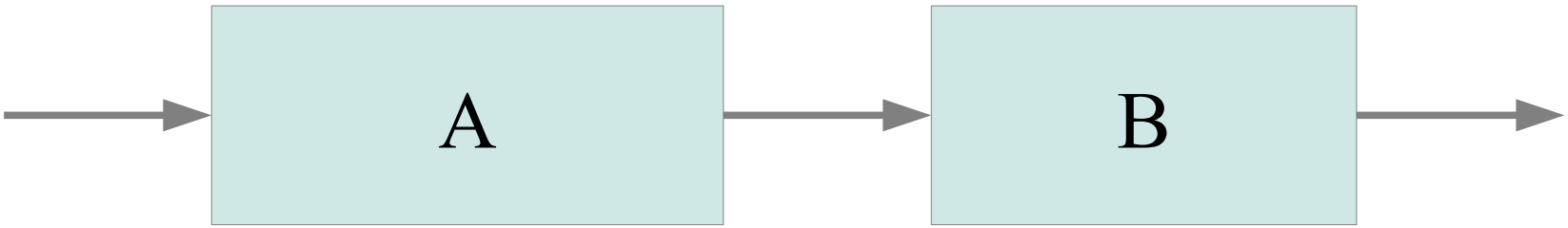
  Може да смятаме че критичния път е възможно най-бавния възможен път за изпълнение на алгоритъма. Това време се нарича още период (span)

- T_1 – Време за изпълнение на 1 процесор

  Без паралелни изпълнения, независимо дали програмата е паралелизирана или не. Това време се нарича още работа (work)

$$T_P \geq T_1/P \qquad T_P \geq T_\infty$$

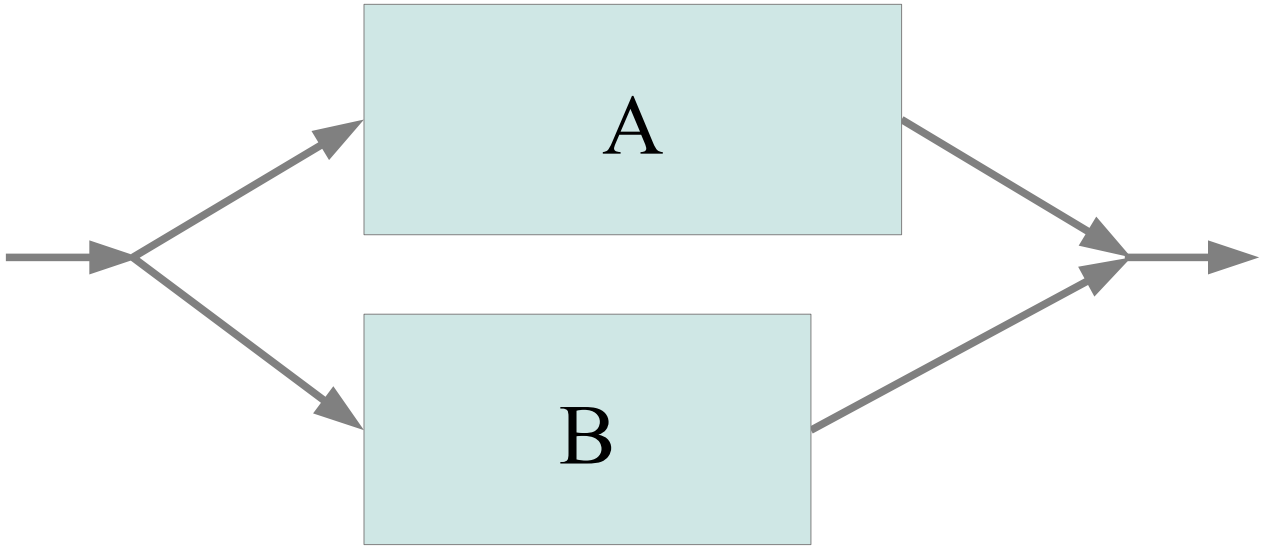
Времена на паралелен алгоритъм



$$T_1(A \quad B) = T_1(A) + T_1(B)$$

$$T_\infty(A \quad B) = T_\infty(A) + T_\infty(B)$$

Времена на паралелен алгоритъм



$T_1(A \quad B) = T_1(A) + T_1(B)$

$T_\infty(A \quad B) = \max(T_\infty(A), T_\infty(B))$

Фактор на ускорение

Фактор на ускорение на P процесора се нарича

$$T_1 / T_P$$

- Линейно при $T_1/T_P = k*P$
- Перфектно линейно при $T_1/T_P = P$
- Супер линейно при $T_1/T_P > P$

Невъзможно? На практика е възможно поради наличие на Кеш паметта и други фактори...

Паралелизъм

Коефициент на паралелизъм се нарича

$$T_1 / T_\infty$$

- За fib(4) имаме Паралелизъм ≈ 17/8 = 2.125
  Използването на повече от 2 процесора няма да даде съществено подобрение
- За умножение на матрици 1000x1000 имаме Паралелизъм ≈ 10^6
  В зависимост от реализацията паралелизма може да достигне и 10^7
- За един нормален RayTracer (при сцена 1000x1000 пиксела) имаме Паралелизъм ≈ 10^6
  Много груба оценка и то за не рекурсивен RayTracer. На практика може да е и много по-голяма

Пример 2

```
void add(hashtable table, value v) {
  int h = hash(v.key);
  v.next = table[h].next;
  table[h].next = &v;
}
```

Може да възникне проблем при паралелно изпълнение

```
h1 = hash(x.key);
x.next = table[h1].next;

table[h1].next = &x;
```

```
h2 = hash(y.key);

y.next = table[h2].next;

table[h2].next = &y;
```

Какъв е проблемът тук?

Пример 2 – използване на критична секция

```
void add(hashtable table, value v) {
  int h = hash(v.key);
  mutex m;
  m.lock();
  v.next = table[h].next;
  table[h].next = &v;
  m.unlock();
}
```

Критична секция

```
h1 = hash(x.key);
m.lock();
x.next = table[h1].next;
table[h1].next = &x;
m.unlock();
```

```
h2 = hash(y.key);



m.lock();
y.next = table[h2].next;
table[h2].next = &y;
m.unlock();
```

Няма проблем при h1=h2, обаче...
Какъв е проблема при тази реализация на критичната секция?

Пример 2 – по-добро използване на к.с.

```
void add(hashtable table, value v) {
  int h = hash(v.key);
  table[h].mutex.lock();
  v.next = table[h].next;
  table[h].next = &v;
  table[h].mutex.unlock();
}
```

```
h1 = hash(x.key);
table[h1].mutex.lock();
x.next = table[h1].next;
table[h1].next = &x;
table[h1].mutex.unlock();
```

```
h2 = hash(y.key);
table[h2].mutex.lock();
y.next = table[h2].next;
table[h2].next = &y;
table[h2].mutex.unlock();
```

Така е по-добре, защото няма заключване при h1≠h2

Извод

Внимателно с критичните секции!

Други опасности и проблеми

- Мъртва хватка (deadlock)
  Два или повече процеси взаимно се изчакват за достъп до общи ресурси, като всички попадат в изчакващо състояние
- Жива хватка (livelock)
  Два или повече процеси взаимно се изчакват за достъп до общи ресурси, като за разлика от мъртвата хватка те не са в изчакващо състояние, а активно се опитват да получат ресурсите без да извършват никаква друга полезна работа
- Конвоиране (convoying)
  За разлика от мъртвата хватка процесите не се блокират и работят, но непрекъснато се изчакват един друг, като въпреки че вършат и полезна работа, то тя е много малко, което води до драстично падане на производителността. Например може да се получи, че процесите чакат един от тях, а той изчаква нов квант време за да продължи работата си
- Съперничество (contention)
  Проблеми с общността на заключването: Ред вместо таблица, клетка вместо ред...
- Допълнително време (overhead)
  Понякога времето за изпълнение на организационните дейности по паралелизация, критични секции и други е прекалено много в сравнение с алгоритъма
- Трудни за отриване и отстраняване на грешки

Решения

- Използване на свободни от заключване структури
- Използване на високо паралелни алгоритми
- Transactional memory
- Read-Copy-Update (RCU)
- Map-Reduce алгоритми
- И много други...