8. Паралелни алгоритми за матрици, изрази и сортиране

Васил Георгиев

ci.fmi.uni-sofia.bg/

v.georgiev@fmi.uni-sofia.bg

Съдържание

- Паралелна обработка за
  - префиксни изчисления
  - матрици
  - изрази
  - сортиране

Префиксни изчисления

- дефинират се върху наредено множество от реални компоненти {x1, x2, …, xn} и асоциативна бинарна опреция върху тях ⊗ – напр. събиране и умножение на реални числа, минимум и максимум на две числа, конкатенация на низове и логическите опреции върху два булеви операнда; за простота някъде ⊗ е записана като сума («+») по-надолу, но операцията не е непременно комутативна!)
- префиксните изчисления (или суми) са стойностите (8.3.1)

  S1 = x1,
  S2 = S1 + x2,
  ...
  Sn = Sn-1 + xn
- префиксните изчисления се прилагат при изчисление на полиноми, CAD системи, диспечеризация, сливане на списъци, обработка на графи и др. поради което са микрокодирани като операции в някои специализирани процесори
- префиксните суми се изчисляват паралелно в двоично дърво с дълбочина $d = (\lceil \mathrm{lb} n \rceil + 1)$ за d+1 стъпки към корена и още d стъпки към листата – общо 2d+1 стъпки – 8.3.2

Префиксно изчисление в хиперкуб

- за изчисление на префиксните суми на n елемента в n-процесорен хиперкуб всяка двойка съседни елементи xi и xi-1 се разполага в съответния процесор pi и в края на обработката в него се получава Si
- за целта (8.4):
  - стъпка 1: в p_i $S_i \leftarrow x_{i-1} \otimes x_i$ (i = 1, …, n; “+” представя $\otimes$)
  - стъпка 2: k = 2; p_i чете S_{i-k} от p_{i-k} и $S_i \leftarrow S_{i-k} \otimes S_i$ (i = k+1, …, n)
  - стъпка 3: k = k + k; преход към стъпка 2 (докато k < n)

- псевдокод:

```
ParallelPrefixComput(X, ⊗, S)
step0: /*initialization
       forall i = 1..n do in parallel
         p[i] reads X[i]; S[i] = X[i]
       endfor
step1: forall i = 1..n do in parallel
         S[i] = S[i-1] ⊗ S[i]
       endfor
step2: k = 2
       while (k < n)
         forall i = k..n do in parallel
           S[i] = S[i-k] ⊗ S[i]
         endfor
         k = k +  k
       endwhile
end
```

Матрични изчисления

- произтичат от всички проблеми, които се рашават със средствата на линейната алгебра и някои типични задачи за обработка на матрици се приемат като еталонни алгоритми за изследване на производителността на парелелните системи
- матриците са пълни или плътни (dense) – респ. без или с малък брой нулеви елементи или разредени (sparse) – с малък брой ненулеви елемнти
- паралелните алгоритми се дават във вид за плътни квадратни матрици ($n \times n$), останалите случаи могат да бъдат обобщени
- основния подход за паралелна матрична обработка е декомпозицията – блокова или решетъчна (“checkerboard partitioning”)

Блокова и решетъчна декомпозиция

- блоковата декомпозиция се извършва или само по колони или само по редове; при плътните матрици всеки блок съдържа еднакъв брой колони респ. редове
- броят на блоковете n е желателно да бъде кратен на броя процесори p за по-добро балансиране, като обкновено n = k*p (за скалируемост на алгоритъма)
- разпределението на блоковете по процесори може да бъде групово или циклично – 8.6
- решетъчната декомпозиция се извършва едновременно по колони и по редове с еднаква честота в двете направления

Решетъчна декомпозиция

- с оглед на базовата операция матрично или матрично-векторно умножение, решетъчната декомпозиция може да се изпълни по различни начини в зависимост от съотношението на размера на матриците и векторите (респ. $n \times n$ и $n \times 1$) и броя обработващи процесори:
  - $p > n^3$ – всички умножения на елементите на операндите могат да се извършат едновременно като n копия на матрицата се разполагат последователно на съседни възли и колоната-вектор се репликира също n пъти
  - $p \geq n^2$ – при матрично-векторно умножение всички произведения могат да се извършат паралелно, при матрично-матрично умножение се прилагат следните подходи
  - всеки стълб (или ред) от резултата се обработва паралелно на една стъпка – общо n последователни стъпки
  - за n последователни стъпки се получават събираемите, от които е съставен всеки компонент на резултата
  - $p < n^2$ – матричите се обработват по части като се прилага блокова декомпозиция – по редове за първия операнд и по стълбове – за втория

Матрично-векторно умножение

- $C_{1\times n} = A_{n\times n} * B_{1\times n} \Leftrightarrow c_i = \sum_{j=0,\ n-1} a_{ij} b_j$, където $C_{1\times n} = [c_0, c_1, \ldots, c_{n-1}]^T$, $A_{n\times n} = [a_{ij}]$ и $B_{1\times n} = [b_0, b_1, \ldots, b_{n-1}]^T$
- оценката на последователния алгоритъм е квадратична (ако се приеме умножението на два елемента и добавянето им към текущия векторен елемент като базова операция):

```
Procedure MatrixVector(A, B, C)
begin
  for i = 0 to n−1 do
    begin
      C[i] = 0
      for j = 0 to n−1 do
        C[i] = C[i] +  A[i,j]*B[j]  /* basic operation
      endfor
    endfor
end
```

- паралелното матрично-векторно умножение може да се изпълни както с блокова, така и с решетъчна декомпозиция

Матрично-векторно умножение с блокова декомпозиция

- при n-процесорна архитектура директния подход е всеки възел да зареди ред от матрицата и колоната на вектора и да изчисли съответния елемент на резултата-вектор:

```
Procedure ParallelMatrixVector_Row(A, B, C)
begin
  for i = 0 to n−1 do in parallel
        C[i] = C[i] +  A[i,0:n−1]*B[0:n−1]
  endfor end
```

- при което оценката за всеки процесор и за системата е $O(n)$ (за n процесора резултата е n^2 операции); очевидно най-удобно в този случай е приложението на блокова декомпозиция по редове
- ако $p < n$ се прилага горната схема на блокова декомпозиция по редове (циклична или групова – 8.9); по-добро балансиране (т.е. ефективност) се постига при кратност на отношението $n:p$
- ако $p > n$ и $n:p = k$ за ефективна обработка е необходимо всяка група от k процесора да си разпредели съответен ред от матрицата и част от векторната колона (което е всъщност решетъчна декомпозиция)

Матрично-векторно умножение с решетъчна декомпозиция

- при горните условия за контектста n^2-процесорна архитектура директния подход е да се формира процесорна решетка като всеки възел зареди съответен елемент от матрицата а колоната на вектора се зарежда в първия ред от n процесори
- паралелния алгоритъм се изпълнява в 3 стъпки:
  - стъпка 1: разпространяване на вектора във всички редове на процесорната решетка
  - стъпка 2: локално умножение на двойката елементи на матрицата и вектора
  - стъпка 3: сумиране на елементите на резултата по редове
- по конвенция резултата се разполага в диагоналните процесорни елементи от решетката; в хиперкуб със същия брой процесори обработката е по-бърза поради по-високата валентност на възлите (респ. по-бързото разпространение на междинните резултати
- този алгоритъм може да се приложи както в MIMD, така и в SIMD архитектури

Матрично-матрично умножение

- $C_{n\times n} = A_{n\times n} * B_{n\times n} \Leftrightarrow c_{ij} = \sum_{k=0,\ n-1} a_{ik}b_{kj}$, където $C_{n\times n} = [b_{ij}]$, $A_{n\times n} = [a_{ij}]$ и $B_{n\times n} = [\ b_{ij}]$
- т.е. c_{ij} е продукт от A_i и B_j (съответно ред и колона)
- при умножение на повече от две матрици се използва последователно асоциативността на опрецията (която не е комутативна – т.е. зададния ред не може да се нарушава): $C = C_1C_2\ldots C_n = (..(C_1C_2)C_3)\ldots C_n)$
- псевдокод:

```
Procedure MatrixMatrix(A, B, C)
  for i = 0 to n−1 do
    for j = 0 to n−1 do
      C[i,j] = 0
      for k = 0 to n−1 do
        C[i,j] = C[i,j] + A[i,k]*B[k,j]
      endfor
    endfor
  endfor
end
```

Матрично-матрично умножение в двумерна процесорна решетка

- алгоритъмът се обработва от в $n \times n$ затворена процесорна решетка (SIMD) и стартира със зареждане на елементите на операндите a_{ij} и b_{ij} в процесора p_{ij} – при това само n процесора (по главния диагонал) съдържат двойка елементи за умножение
- паралелния алгоритъм се изпълнява в 3 стъпки:
  - стъпка 1: за комбиниране на подходящите двойки елементи n пъти се извършва ротационно местене на B-елемнтите нагоре и на A-елементите наляво
  - стъпка 2: локално умножение на двойката елементи на матриците
  - стъпка 3: разпространение на локалните междинни резултати към съседните възли наляво и нагоре за - n итерции – след което резултата c_{ij} се съдържа в p_{ij}

Матрично-матрично умножение в двумерна процесорна решетка – код, 8.13

```
Procedure
   ParallelMatrixMatrix_2d(A, B, C)
Step1
 for k = 0 to n−1 do
   for Pij where i,j = 0 to n−1 do in
    parallel
     if i>k then
       A[i−1, j] ← A[i, j] endif
     if j>k then
       B[i, j−1] ← B[i, j] endif
   endfor
 endfor
Step2
```

```
... Step2
 for Pij where i,j = 0 to n−1 do in
                              parallel
   C[i, j] = A[i, j] ∗ B[i, j]
 endfor
Step3
 for k = 0 to n−1 do
   for Pij where i,j = 0 to n−1 do in
    parallel
     A:Pij ← (move_left)A:Pij        /∗ 3.1
     B:Pij ← (move_up)B:Pij          /∗ 3.2
     C[i, j] = C[i, j] +  A[i, j]∗B[i, j]  /∗
                                      3.3
   endfor
 endfor
end
```

Матрично-матрично умножение в тримерна процесорна решетка

- алгоритъмът се обработва от в $n \times n \times n$ процесорна решетка (SIMD) или хиперкуб като $\exists q$: $n = 2^q$ т.е. $p = 2^{3q}$
- при горното условие n = 2, 4, 8 … и номерацията на процесорите има фо́рмата $p_{ijk} = p_x$ за $x = in^2 + jn + k$ (i, j, k = 0, 1, .., n-1 ; x = 0, 1, .., n^3-1), което съответства на номерацията в хиперкуб (8.14)
- ако архитектурата с n^3 процесори е решетка, а не хиперкуб (т.е. валентността на възлите е константа 4, а не $\mathrm{lb}(n^3) = 3q$), този алгоритъм е в сила, но преноса на операнди няма да бъде само между съседни възли
- алгоритъмът изпълнява паралелно n^3 умножения с което обработва n^2 елементи от резултата:

… Матрично-матрично умножение в тримерна процесорна решетка

- стъпка 1: елементите a_{ij} и b_{ij} се зареждат в процесор $p_{(ni+j)}$ (процесори $0 \div n^2$-1)
- стъпка 2: разпространение на операндите до останалите процесори
- стъпка 3: в процесор p_{ijk} $c_{ijk} = a_{ji} * b_{ik}$ (след зареждане на необходимите един или два операнда от съседите)
- стъпка 4: след сумиране $c_{ij} = \sum_{k=0,\ n-1} c_{ijk}$ резултата c_{ij} се намира в процесор $p_{(ni+j)}$
- пример за $n = 2$ (8.15)

Система линейни уравнения

- системата линейни уравнения $\sum_{i,j=1,n} a_{ij}x_j = b_i$ има матричното представяне $A_{n\times n}\, x_{1\times n} = b_{1\times n}$ с решение $x = A^{-1}b$ – при условие, че A е неизродена матрица (т.е. редовете и стълбовете ѝ не са линейно зависими);
- [***диагонална*** е матрица с ненулеви елементи само по главния диагонал; когато ненулевите елементи на диагоналната матрица са само единици, тя е ***единична*** матрица I_n като $AA^{-1} = A^{-1}A = I_n$; ***тридиагонална*** е матрица с ненулеви елементи само по главния и двата съседни диагонала (т.е. за които $|i - j| \leq 1$); ***долно-триъглна*** е матрица с ненулеви елементи само над главния диагонал и обратно – ***горно-триъгълна*** е матрица с ненулеви елементи само под главния диагонал]
- при представяне на коефициентите (т.е. елементите на матриците) с плаваща запетая решението $A^{-1}b$ често поражда числова нестабилност; на практика се прилага метода на LU-декомпозицията (в математиката – метод на Гаус), който числово е по-стабилен и се обработва около три пъти по-бързо

Решаване на система линейни уравнения

- идеята на различните методи за решаване на СЛУ е привеждане на разширената матрицата $Ab_{n \times n+1}$ към горно-триъгълна форма (съответстваща на редуциране на променливите) чрез елементарни опареции по редове така че под главния диагонал остават само нулеви елемнети, а диагоналните елементи са единици – фаза елиминиране (forward elimination); след това се извършва фазата заместване (back-substitution), при която *A* се трансформира в единична матрица, а в стълба *b* се съдържа решението на системата

LU-декомпозиция

- това е приложен метод за генериране на $n \times n$ матриците L и U, за които е в сила:
  - L е единична долно-триъгълна
  - U е горно-триъгълна
- фазата елиминиране започва с добавяне на подходящи изрази към всички уравнения с изключение на първото, с което се елиминира първата променлива; по същия начин се елемиминират последователно променливите от следващите уравнения – в общия случай за да се елиминира *i*-тата променлива от *j*-тото уравнение най напред се умножава *i*-тото уравнение с a_{ji}/a_{ii} и полученото еквивалентно уравнение се изважда от *j*-тото уравнение
- определя се базов ред (уравнение), който се използва за нулиране на елементите под главния диагонал в колона *i*, (pivot)

LU-декомпозиция: псевдокод (тук A е разширената матрица $Ab_{n\times n+1}$)

- сложността на този последователен алгоритъм в двете фази е с кубична $O(n^3)$, тъй като:
  - елиминиране – за всеки ред i цикълът по k се изпълнява $(n – i + 2)$ пъти, а вложеният цикъл по j – $(n – i)$ пъти т.е. общо за фазата $\sum_{i=1,\, n} (n – i + 2)(n – i)$ – което дава линейна оценка
  - заместването е с квадратична оценка

```
Procedure GaussianElimination(A)
begin                    /* Forward elimination
  for i = 0 to n do
    begin
    max = i
    for j = i + 1 to n do
      if(abs(A[j, i])>abs((A[max, i])) then max = j
    for k = i to n + 1 do
      t = A[i, k]; A[i, k] = A[max, k]; A[max, k] = t
    for j = i + 1 to n do
      for k = n + 1 to n do
        A[j, k] = A[j, k] – A[i, k]*A[j, i]/A[i, i]
    end
end
begin          /* back-substitution
for j = n to 1 do
  t = 0
  for k = j + 1 to n do
    t = t + A[j, k]*X[k]
  X[j] = (A[j, n+1] – t)/A[j, j]
end
```

Паралелни версии на LU-декомпозицията

- паралелната обработка на двете фази от LU декомпозицията може да се извърши с разпределяне на алгоритъма между процесорите по редове или колони на матрицата $Ab_{n\times n+1}$
- пример – когато размера на системата е от порядъка на *n*, фазата заместване при паралелна обработка по колони може да се извърши в следните (*n* -1) стъпки:
  - стъпка 1 – вход долно-триъгълна матрица *A* и вектор *B* – напр. 8.20 за $n = 4$ – $(n - 1)$ процесора обработват паралелно изразите от вида $b_i^{(1)} = b_i + a_{i1}x_1$, $i = 2, \ldots, n$ и $x_1 = b_1$
  - стъпка 2 – $(n - 2)$ процесора обработват паралелно изразите от вида $b_i^{(2)} = b_i^{(1)} + a_{i2}x_2$, $i = 3, \ldots, n$ и x_1, x_2 са известни
  - стъпка *k* - $(n - k)$ процесора обработват паралелно изразите от вида $b_i^{(k)} = b_i^{(k-1)} + a_{ik}x_k$, $i = k+1, \ldots, n$ и $x_1, x_2, \ldots, x_k$ са известни

Балансиране на LU-декомпозицията

- броят на обработващите процесори намалява с изпълнението на стъпките, което води до по-ниска ефективност на обработката; този ефект се наблюдава при повечето методи за решаване на СЛУ
- по принцип разделянето на проблема по редове (уравнения) е свързано с определяне на базовия ред и предаване на неговите параметри до останелите процесори, след което всеки процесор обработва един (или повече) от редовете
- аналогична обработка може да се извърши и с разделяне по колони, но в повечето случаи обработката по редове постига по-добро бързодействие

Аритметични изрази

- състоят се от оператори и операнди със скоби за явно задаване на реда на операциите
- при синтактичния разбор (parsing) изразите се преобразуват в дърво; когато операциите в тях са бинарни – и дървото е бинарно – 8.22
- изчислението на израза става рекурсивно по неговото дърво, което логически е еквивалентно на задача за обхождане на бинарно дърво
- елементарни изрази са тези, в които всеки операнд (променлива) участва само веднъж
- достатъчни условия E да е елементарен израз са:
  - $E = x_i$; x_i е променлива
  - $E = \otimes G$; G е прост израз и $\otimes \in \{+, -\}$
  - $E = G \otimes H$; G, H са прости без общи операнди и $\otimes \in \{+, -, *, /\}$
- пример: $E = x_1 * x_2 * x_3$ е елементарен, но $H = x_1^2$ не е
- еквивалентни изрази са тези, които приемат еднакъв набор аргументи (по брой и тип) и връщат един и същ резултат за всеки набор от стойности на тези аргументи – пример: $E_S = (x_1 x_2 + x_3)x_4 + x_5$; $E_P = x_1 x_2 x_4 + x_3 x_4 + x_5$

Паралелна обработка на аритметични изрази

- паралелизмът при решаването на изрази се задава с дървото на разбора, а при потоковите архитектури – с графа на зависимостите
- еквивалентните изрази могат да имат различни дървета на представяне и съответно различна степен на паралелизъм – напр. E_S и E_P (от 1.) са подходящи съответно за последователна и паралелна обработка и имат различно представяне с дърво на разбора и различен паралелизъм – 8.23.1
- за съпоставка на песледователната и паралелната обработка на един израз (респ. на еквивалентни изрази) се използват следните критерии (8.23.2):
  - брой стъпки (последователни или паралелни операции) за решаване на израза
  - брой процесори – т.е. ниво на паралелизма (при MIMD архитектура)
  - общ брой операции

Аритметични изрази в SIMD и MIMD

- допълнителна характеристика за израза е възможноста за обработката от SIMD архитектура, при която паралелно може да се изпълнява само една елементарна операция (от всички процесорни елементи) – пример: еквивалентните изрази
  - $E_0 = (((((x_1*x_2)+ x_3)*x_4) + x_5)*x_6) + x_7$;
  - $E_1 = ((((x_1*x_2)*(x_4*x_6)) + (x_3*(x_4*x_6))) + ((x_5*x_6) + x_7))$ и
  - $E_2 = ((((x_1*x_2) + x_3)*(x_4*x_6)) + ((x_5*x_6) + x_7))$,

  са с различен паралелизъм и сложност (брой стъпки) по отношение на SISD, SIMD и MIMD архитектура – (8.24)
- паралелната обработка на даден израз по принцип се извършва като за всеки връх на дървото се планира процесор:

```
repeat
  for each vertex x do in parallel
    if (children(x) known) then
      compute x
      remove children from the tree  endif  endfor
until only root left
```

- времевата сложноста на такъв алгоритъм е $O(\mathrm{lb}n)$ където n е броя операции и процесори, а стойноста на обработката е $O(n\mathrm{lb}n)$
- поради зависимостта по данни броя процесори може да се намали без да се увеличи времето за решаване на израза – средностатистически $p_{opt} = n/\mathrm{lb}n$; в този случай стойността е $O(n)$

Сортиране

- сортирането на структури от елементи по техен ключ (стойност) е задача, която се изпълнява от компилатори, редактори, системни модули за управление на паметта и процесите и много приложения
- сортирането се разграничава на вътрешно и външно, като при второто само част от сортираните елементи се разполагат в основната памет, а останалите са във външната памет
- сортирането се базира на разделяне на елементите по групи и сравняването им в рамките на групата (в крайна сметка сравняването им по двойки), така че има две последователни фази – разделяне и сливане
- фазата разделяне присъства винаги тъй като операцията сравнение-размяна е бинарна, следователно последователността за сортиране трябва да се раздели по двойки операнди; допълнително предимство на разделянето е, че обработката по части позволява паралелизъм

Методи за сортиране

- два основни метода се прилагат от различни сортиращи алгоритми:
  - сливане – последователността за сортиране се разделя на две равни по размер части, които на свой ред се сортират рекурсивно; след това двете сортирани части се сливат – при този метод сравняването (и избора) на елементите става във втората фаза – сливането («*easy split / hard join*»)
  - разделяне – последователността за сортиране се разделя на две равни по размер части като всеки елемент от първата е по-малък от кой да е елемент от втората; процеса на разделяне продължава рекурсивно за тези части до изчерпване, след което подредените елементи се сливат в сортирана последователност – при този метод сравняването (и избора) на елементите става в първата фаза – разделянето («*hard split / easy join*»)
- ефективността и бързодействието на паралелните версии на сортиращите алгоритми зависят значително от използваната архитектура

Представяне на сортирането с мрежи

- сортирането се представя с мрежи или графи, от които лесно се извлича топологията и процесите на паралелното сортиране
- сортиращите мрежи са комбинация от компаратори – логически устройства, които извършват операцията сравнение-размяна (8.27)
- формално компараторът е четириполюсник – устройство с два входа $I_{1,2}$ и два изхода $O_{1,2}$ – които имат следните свойства:
  - $O_1 = \min(I_1, I_2)$
  - $O_1 = \max(I_1, I_2)$
- компаратори, които нямат общи входове-изходи, могат да функционират паралелно
- компараторите се свързват в компараторни каскади (comparator stages), като изход[ите] на един компаратор е/са вход[ове] на следващ – функционирането на компараторите в каскада е последователно

Сортиращи мрежи

- формално сортиращите мрежи са насочени графи със следните свойства (8.28.1):
  - върховете, от които има само излизаща дъга (ребро) са входове, а тези към които има само влизаща дъга са изходи на графа
  - останалите върхове (освен входовете и изходите) са компаратори
  - изходите на сортиращата мрежа съдържат сортирана последователност от стойностите, които се записват във входовете – т.е. сортиращите мрежи се състоят от компараторни каскади
- с помощтта на сортиращи мрежи алгоритмите за сортиране могат да се проектират систематично с формални средства
- пример (8.28.2): мрежата на сортирането «мехурче» се състои от $n(n-1)/2$ компаратори в $(2n-3)$ каскади, паралелизмът на алгоритъма е $(n-1)$ и при наличие на такъв брой процесори алгоритъмът ще се изпълни за $(2n-3)$ стъпки
- сливаща мрежа е такава сортираща мрежа, на която входовете се делят на две [наредени] равномощни множества и ако на всяко от двете множества входове се подаде сортирана последователност, на изходите се получава общата сортирана последователност

Представяне на сортирането с графи

- сортирането може да се представи и с графи, които обикновено са бинарни дървета, конструирани по следния начин (8.29.1):
  - листата на дървото са входове, в които се разполага несортираната последователност
  - вътрешните върхове изпълняват операцията сравнение-размяна върху последователностите, които се съдържат в техните наследници
- тъй като сортирането има две фази – разделяне и сливане – които се изпълняват последователно в този ред, понякога то се представя като две свързани бинарни дървета (за двете фази – 8.29.2) – или еквивалентно – като едно дърво, на което дъгите са двупосочни и фазата разделяне съответства на движение към листата, а фазата сливане – на движение към корена
- от сортиращото дърво лесно се извлича паралелизма на обработката: върховете от едно ниво могат да се изпълнят паралелно от различни процесори (което съответства на обработка в дълбочина)

Сортиране «четни-нечетни»

- на практика се прилагат предимно алгоритми от класа сортиране чрез сливане; при паралелната обработка най-често се реализира сортиране «четни-нечетни» и битонично сортиране (които са от класа чрез сливане) – причина за което е не само добрите оценки на тези алгоритми, но и това, че схемите (последователностите) на сравнения при тях не зависят от контекста
- базовата версия на сортирането «четни-нечетни» (Odd-even *transposition* sort) извършва операцията сравнение-размяна последователно в двуфазни итерации:
- в първата фаза се сравняват и разменят само четните елементи и техните [нечетни] съседи със следващ по-голям индекс т.е. [*i*: *i*+1]
- във втората фаза на итерацията сравненията са по алтернативните нечетни индекси

Сортираща мрежа odd-even

- сортиращата мрежа на базовия алгоритъм за *n* входни елементи се състои от *n* компараторни каскади; всята каскада се състои от (n-1) паралелно работещите компаратори [*i*: *i*+1] – съответно за четните и за нечетните елементи (8.31)
- броят на компараторите е $n(n-1)/2$
- предимството на сортирането «четни-нечетни» (освен просототата) е запазване на принципа за локалност на опрециите сравняване-размяна и също скалируемостта на алгоритъма и балансиране на операциите между процесорите в рамките на всяка итерация, но ефективността от приложението на n^2 компаратори е ниска
- в паралелна система отделните итерации могат да се изпълняват последователно от *n* процесора
- *Упр: да се сметне ефективността и паралелизма*

Сортиране «четни-нечетни» чрез сливане

- този алгоритъм (Odd-even *merge* sort) изпълнява сортиращо сливане на две *сортирани* последователности с еднакъв размер и рекурсивно разделяне на по-късите последователности по четни и нечетни индекси
- псевдокод:

```
Algorithm OddEven(A, B, S)     /* A,B sorted subsequences of S
  begin
  if A, B are of length 1 then Compare-Exchange-Merge
  else
  begin
    form A_odd, B_odd, A_even, B_even              /* step 1
    compute in parallel
      OddEven(A_odd, B_odd, S_odd)                /* step 2
      OddEven(A_even, B_even, S_even)
      S_odd-even = Merge(S_odd, S_even)           /* step 3
      S_odd-even = OddEvenInterchange(S_odd-even) /* step 4
  end
  endif   end
```

Пример: Odd-Even merge

- пример: A = {2, 6,10, 15} и B = {3, 4, 5, 8} са двете равни по размер сортирани последователности за сливане в S (сортираща мрежа: 8.33):
  - стъпка 1: A_{odd} = {2, 10}, B_{odd} = {3, 5}, A_{even} = {6,15}, B_{even} = {4, 8};
  - стъпка 2: S_{odd} = {2, 3, 5, 10}, S_{even} = {4, 6, 8, 15};
  - стъпка 3: $S_{odd\text{-}even}$ = {2, **4**, **3**, **6**, **5**, **8**, **10**, 15};
  - стъпка 4: $S_{odd\text{-}even}$ = {2, 3, 4, 5, 6, 8, 10, 15};
- дълбочината на рекурсия е логаритмична; при всяка итерация операциите сливане и сравняване-размяна се изпълняват $n/2$ пъти, върху n процесора те се изпълняват за 1 стъпка – следователно времето за обработка остава логаритмично, а стойността на обработката е $O(n\text{lb}n)$ – която е оптимална в сравнение със всеки последователен сортиращ алгоритъм

Паралелен Odd-Even merge

```
Procedure OddEvenMerge(L[1:n])
  Model: n-processor PRAM
  Input: L[1:n]; n=2**k, L[1:n/2] and L[n/2+1:n] sorted
  Output: L[1:n] sorted
  if (n=2) then
    if (L[1]>L[2]) then exchange(L[1], L[2]) endif
  else
    OddEvenSplit(L[1:n], Odd[1:n/2], Even[1:n/2] /*separate list elements
                                                  /*of odd and even indices
    OddEvenMerge(Odd[1:n/2])                      /*recursive sorting
    OddEvenMerge(Even[1:n/2])
    for i = 1 to n/2 do in parallel
      L[2i-1] = odd[i]
      L[2i] = even[i]
    end in parallel
    for i = 1 to n/2 do in parallel
      if (L[2i]>L[2i+1]) then exchange(L[2i]>L[2i+1]) endif
    end in parallel
  endif
end OddEvenMerge
```