Припреме за лабораторијске вежбе из предмета Системска програмска подршка у реалном времену II
- 2017-2018/ Вежба 10 -

Област вежби: *Паралелно програмирање*

ТВВ, ТУТОРИЈАЛ III

РАСПОРЕЂИВАЧ ЗАДАТАКА

Садржај

- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Увод

- Распоређивач задатака је основа шаблона петљи.
- Шаблони петљи (нпр. `parallel_for`) скривају сложеност распоређивача.
- Када проблем (алгоритам) не одговара шаблону петљи високог нивоа, може се директно користити распоређивач задатака.
- Могу се правити нови шаблони високог нивоа.

Садржај

- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Програмирање засновано на задацима (1/7)

- Много је боље програм формулисати помоћу логичких задатака, а не нити, из више разлога:
  - Увођење паралелизма коришћењем расположивих ресурса,
  - Већа брзина покретања и уништавања задатка,
  - Ефикаснија процена,
  - Боље уравнотежење оптерећења,
  - Размишљање на високом нивоу.

Програмирање засновано на задацима (2/7)

- Нити направљене са доступним пакетом би биле логичке нити које се пресликавају на физичке нити (тј. језгра) процесора.
- За рачуне који не зависе од спољних догађаја, највећа ефикасност се постиже када се тачно једна логичка нит извршава унутар једне физичке.
- Могући проблеми:
  - Логичких нити има мање од физичких,
  - Логичких нити има више од физичких.

Програмирање засновано на задацима (3/7)

- Други проблем је већи јер доводи до извршавања логичких нити у временским исечцима (*time slice*), што захтева додатно време извршења.
- Распоређивач покушава да заобиђе овај проблем стварајући једну логичку нит по физичкој нити, пресликавајући задатке на логичке нити и узимајући у обзир сметње са другим нитима истог или другог процеса.

Програмирање засновано на задацима (4/7)

- Важна предност задатака у односу на нити је да су много бољи у погледу времена покретања и уништавања (на Linux OS-у до 18 пута брже, на Windows-у и до 100 пута):
  - Нити, за разлику од TBB задатака, имају своју локалну копију многих ресурса (стање регистара, магацинска меморија, итд), па чак и свој идентификатор процеса (*process id*),
  - TBB задатак је најчешће само мала рутина.

Програмирање засновано на задацима (5/7)

- ТВВ задаци су такође ефикасни јер је распоређивање неправедно (*unfair*).
- Најчешће се временски одсечци додељују редом у круг (праведно), јер је то најбезбеднија стратегија која се може предузети без спознаје вишег нивоа организације програма.
- Како ТВВ има информације о вишем нивоу програма, може и да жртвује праведност ради ефикасности.

Програмирање засновано на задацима (6/7)

- Распоређивач ради уравнотожења оптерећења (load balancing):
  - Поред коришћења одговарајућег броја нити, потребно је равномерно распоредити задатке на расположиве нити.
  - Потребно је да програм буде издељен на довољно мале задатке да би распоређивач коректно доделио задатак нити у циљу уравнотожења оптерећења.

Савет: Пројектовати програм тако да ствара много више задатака него што има нити, и да препусти распоређивачу пресликавање задатака на нити.

Програмирање засновано на задацима (7/7)

- Основна предност задатака у односу на нити је могућност размишљања на вишем нивоу.
- Са нитима се мора размишљати на ниском нивоу физичких нити да би се постигла висока ефикасност (једна логичка нит по физичкој).
- Такође, мора се радити са доста грубом поделом нити.
- Са задацима је могуће концентрисати се на логичке зависности задатака, а распоређивање препустити самом распоређивачу.

Садржај

- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Када не користити програмирање засновано на задацима (1/2)

- Распоређивач задатака је намењен за алгоритме високих перформанси који се састоје од искључиво неблокирајућих задатака.
- Може бити користан и када се задатак понекад (ретко) блокира.
- Уколико долази до честог блокирања нити (чекање на улаз/излаз, семафор и сл.), долази до губитка перформанси (док је нит блокирана, не ради ни на једном задатку).

Када не користити програмирање засновано на задацима (2/2)

- Уколико нити чекају, програм се неће добро извршавати без обзира на број нити које постоје.
- Ако постоје блокирајући задаци, најбоље је користити нити.
- Распоређивач задатака у потпуности подржава комбиновање нити са ТВВ задацима.

Садржај

- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Пример – Фибоначијеви бројеви (1/9)

- Пример рачунања *n*-тог Фибоначијевог броја.
- Користи неефикасан начин рачунања, али демонстрира основе библиотеке са задацима, користећи једноставан образац рекурзивних задатака.
- ТВВ образац рекурзивних задатака омогућава стварање већег броја задатака, што је услов за скалабилност убрзања у програмирању заснованом на задацима.

Пример – Фибоначијеви бројеви (2/9)

- Секвенцијални код:

```
long SerialFib( long n ) {
    if( n<2 )
        return n;
    else
        return SerialFib(n-1)+SerialFib(n-2);
}
```

- Код највишег нивоа за паралелну верзију засновану на задацима:

```
long ParallelFib( long n ) {
    long sum;
    FibTask& a = *new(task::allocate_root())
FibTask(n,&sum);
    task::spawn_root_and_wait(a);
    return sum;
}
```

Заузимање простора за задатак преклопљеним оператором `new` и методом `task::allocate_root`

Задатак

Прављење и покретање задатка

Конструктор

Пример – Фибоначијеви бројеви (3/9)

- `_root` суфикс у имену методе `task::allocate_root` наглашава да створени задатак нема претка, већ је коренски задатак у стаблу.
- Простор за задатак се мора заузети посебним методама да би се простор ефикасно употребљавао и када се дати задатак заврши.
- Задатак `FibTask` створен конструктором `FibTask(n,&sum)` имплицитно позваним оператором `new`, рачуна *n*-ти Фибоначијев број и смешта га у `*sum`, након покретања методом `task::spawn_root_and_wait`.

Пример – Фибоначијеви бројеви (4/9)

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
        n(n_), sum(sum_)
    {}
    task* execute() {  // Overrides virtual function task::execute
        if( n<CutOff ) {
            *sum = SerialFib(n);
        } else {
            long x, y;
            FibTask& a = *new( allocate_child() ) FibTask(n-2,&x);
            FibTask& b = *new( allocate_child() ) FibTask(n-1,&y);
            // Set ref_count to "two children plus one for the wait".
            set_ref_count(3);
            // Start b running.
            spawn( b );
            // Start a running and wait for all children (a and b).
            spawn_and_wait_for_all(a);
            // Do the sum
            *sum = x+y;
        }
        return NULL;
    }
};
```

Пример – Фибоначијеви бројеви (5/9)

- Овакав задатак је описан релативно великим кодом у односу на `SerialFib`, јер изражава паралелизам заснован на задацима, без помоћи икаквих додатака на стандардни C++.
- Свака класа (структура) која представља задатак, па тако и `FibTask`, наслеђује класу `task`.
- Поља `n` и `sum` чувају улазну вредност односно показивач на излазну вредност. То су копије аргумената прослеђених конструктору `FibTask`.

Пример – Фибоначијеви бројеви (6/9)

- Метода `execute` представља сам рачун.
- Сваки задатак мора да има дефиницију методе `execute` која преклапа чисто виртуелну методу `task::execute`.
- Њена дефиниција треба да обавља посао задатка и да врати `NULL` или показивач на следећи задатак који треба покренути.

Пример – Фибоначијеви бројеви (7/9)

- Метода `FibTask::execute` ради следеће:
  - Проверава да ли је `n` толико мало да је серијско извршење брже (`CutOff` је у овом примеру 16, одређено експерименталном методом). Прелазак на секвенцијални алгоритам у случају да је проблем мали је карактеристика шаблона паралелизма *подели-и-завладај*.
  - У `else` грани се праве два задатка потомка, који рачунају (n-1)-ви и (n-2)-ги Фибоначијев број. Овде се користи наслеђена метода allocate_child() за заузимање простора за задатак јер овде задатак прави задатак-потомак.
  - Позива `set_ref_count(3)` обавезно пре мрешћења (*spawn*) било ког потомка. Број 3 представља 2 задатка потомка и додатни имплицитно уведен задатак за потребе методе `spawn_and_wait_for_all`.

Пример – Фибоначијеви бројеви (8/9)

- Ствара два задатка потомка. Стварање задатка даје знак распоређивачу да може да га покрене у било ком тренутку, могуће и паралелно са другим задацима. Прво стварање потомка, методом `spawn`, се завршава одмах, без чекања на завршетак извршења задатка потомка.
- Следеће стварање, методом `spawn_and_wait_for_all`, доводи до чекања задатка родитеља да се заврше сви претходно покренути задаци потомци.
- Након завршетка оба задатка потомка, задатак предак рачуна `x+y` и резултат смешта у `*sum`.

Пример – Фибоначијеви бројеви (9/9)

- На први поглед се чини да је паралелизам ограничен јер се стварају само два задатка потомка, али трик је у рекурзивном паралелизму:
  - Сваки од два задатка потомка ствара нова два задатка потомка и тако док није `n<Cutoff`.
  - Предност распоређивача задатака је што тако настали потенцијални паралелизам претвара у стварни паралелизам на врло ефикасан начин – бирајући задатке које покреће тако да физичке нити одржава заузетим уз релативно малу замену контекста нити.

Садржај

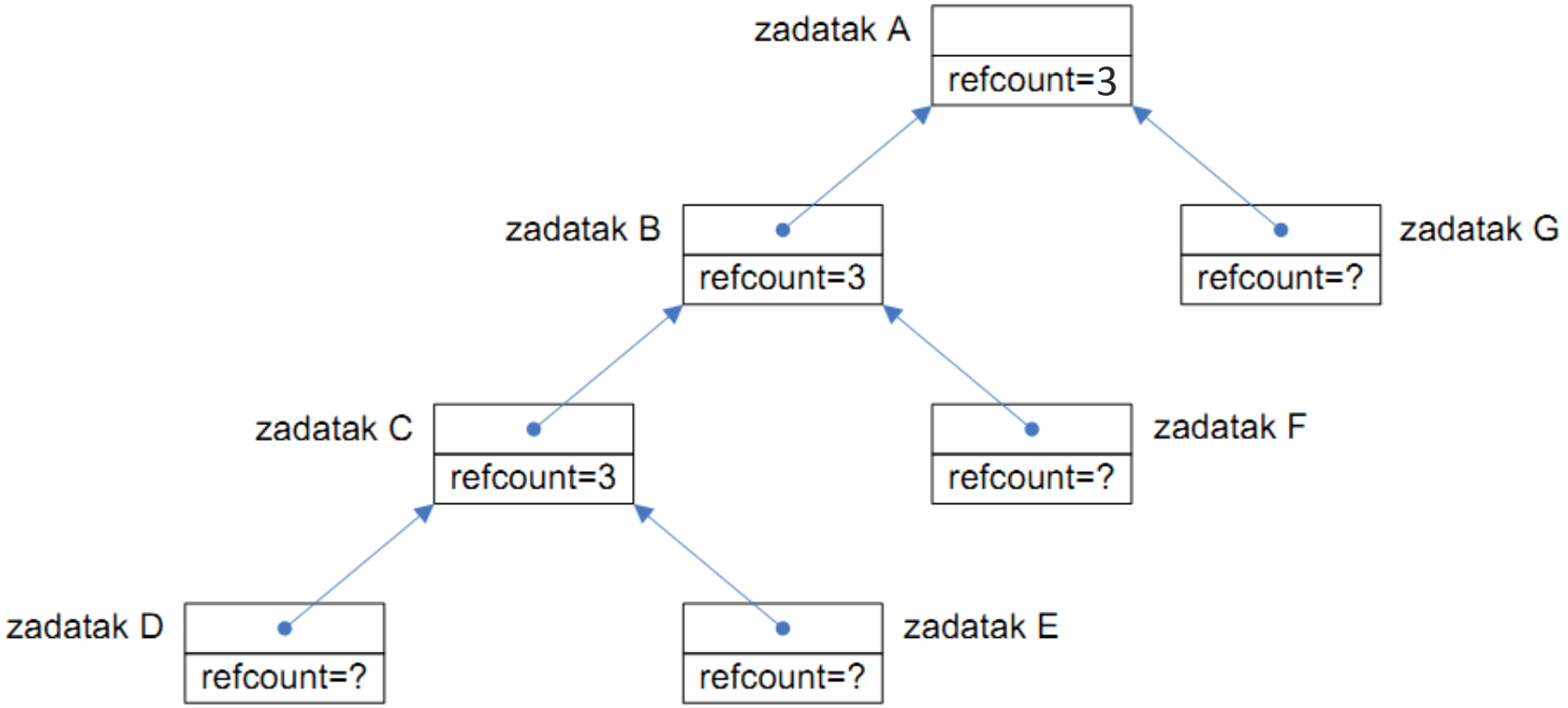
- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Како ради распоређивач (1/5)

- Распоређивач задатака формира граф задатака
  - Усмерени граф где је сваки чвор графа један задатак.
  - Сваки задатак показује на свог следбеника, што је следећи задатак који га чека да се заврши, или `NULL`.
  - Метода `task::parent()` даје приступ читања показивачу на следбеника.
  - Сваки задатак има `refcount` који означава број задатака којима је он следбеник.
  - Граф се развија у времену.

Како ради распоређивач (2/5)

- Пример графа (Фибоначијеви бројеви)



Како ради распоређивач (3/5)

- Под претпоставком да је стабло задатака коначно, распоређивач узима за приоритет при покретању задатака ***дубину***, што је најбоље за секвенцијално извршавање из два разлога:
  - *Погодак када је скривена меморија ажурна*
  - *Смањивање простора*
- Извршавање по ***ширини*** уноси вишеструке проблеме са утрошком меморије, али би знатно повећало паралелизам када би постојао неограничен број физичких нити. Овако се користи само у довољној мери да се сваки процесор (језгро) одржи заузетим.

Како ради распоређивач (4/5)

- Распоређивач уводи комбинацију извршавања по ширини и дубини:
  - Свака нит има свој ред са два краја (*deque*) задатака који су спремни за покретање.
  - Када нит створи задатак, додаје га на крај свог реда.

- Када нит учествује у развијању стабла, она извршава задатак одабран по првом од правила (редом):
  - Узми задатак који је вратила метода `execute` претходног задатка (осим уколико је вратила `NULL`). (по дубини)
  - Узми задатак са дна свог реда (осим уколико је ред празан). (по дубини)
  - Преузми задатак са врха другог, случајно изабраног, реда (који није празан). (по ширини, доводи до паралелизма)

Како ради распоређивач (5/5)

- Бирање и узимање задатка је увек аутоматско.
- Додавање задатка у ред може бити имплицитно или експлицитно:
  - Нит увек додаје задатак у свој ред, никад у други.
  - Само операција преузимања (*steal*) може да пребаци задатак створен од једне нити у ред друге нити.
  - Постоје три узрока због којих нит додаје задатак у свој ред

Садржај

- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Празан задатак - Empty Task (1/4)

- У случају да је потребан задатак који не ради ништа, осим што чека на своје потомке да се заврше.
- Заглавље task.h дефинише класу `empty_task` која се може користити у те сврхе.

Празан задатак - Пример (2/4)

```
//! Start up a SideShow task.
//! Return pointer to empty (dummy) task that acts as parent of the SideShow.
tbb::empty_task* StartSideShow( ) {
    tbb::empty_task* parent = new( tbb::task::allocate_root( ) ) tbb::empty_task;
    // 2 = 1 for SideShow and 1 for the wait
    parent->set_ref_count(2);
    SideShow* s = new( parent->allocate_child( ) ) SideShow;
    printf(" [DEBUG][%s] ref_count = %d\n", __FUNCTION__,parent->ref_count()); //!
    __TBB_ASSERT(parent->ref_count()==2, NULL );
    parent->spawn(*s);
    return parent;
}

//! Wait for SideShow task. Argument is empty parent of the SideShow.
void WaitForSideShow( tbb::empty_task* parent ) {
    //continue execution when ref_count reach 1
    parent->wait_for_all( );
    __TBB_ASSERT(parent->ref_count()==0, NULL );
    printf(" [DEBUG][%s] ref_count = %d\n", __FUNCTION__,parent->ref_count()); //!
    // parent not actually run, so we need to destroy it explicitly.
    // (If you forget this line, the debug version of tbb reports a task leak.)
    parent->destroy(*parent);
}
```

Заузимање простора за празан задатак

Заузимање простора за задатак потомак

Прављење и покретање задатка потомка

Чекање да се задатак потомак заврши

Ослобађање ресурса празног задатка

Празан задатак - Пример (3/4)

```
//! Some busywork
void TwiddleThumbs( const char * message, int n ) {
    for( int i=0; i<n; ++i ) {
        printf(" %s: i=%d\n",message,i);
        static volatile int x;
        for( int j=0; j<20000000; ++j ){
            ++x;
         }
     }
}

//! SideShow task
class SideShow: public tbb::task {
    tbb::task* execute( ) {
        TwiddleThumbs("Sideshow task",4);
        return NULL;
    }
};
```

Функција обраде (симулација)

Функција задатка потомка

Празан задатак - Пример (4/4)

```
//! Optional command-line argument is number of threads to use. Default is 2.
int main( int argc, char* argv[] ) {
    tbb::task_scheduler_init init( argc>1 ? strtol(argv[1],0,0) : 2 );
    // Loop over n tests various cases where SideShow/Main finish twiddling first.
    for( int n=3; n<=5; ++n ) {
    printf("\ntest with n=%d\n",n);

    // Start up a Sideshow task
    tbb::empty_task* e = StartSideShow( );

    // Do some useful work
    TwiddleThumbs("master",n);

    // Wait for Sideshow task to complete
    WaitForSideShow(e);
    }
    return 0;
}
```

Покретање новог задатка

Настави извршење – преузимање неког другог посла

Чекање на завршетак задатка

Садржај

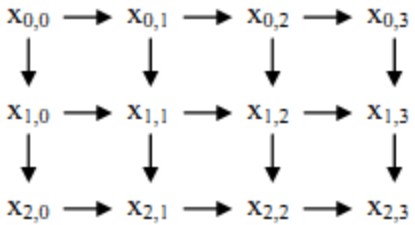
- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Општи ациклични граф задатака (1/4)

- Граф задатака не мора бити само у облику стабла, где сваки чвор има тачно једног наредног, `task::parent()`, који чека да се он заврши.
- Да би омогућио сложеније графове, *TBB* има методе за непосредно руковање бројем референци конкретног задатка.

Општи ациклични граф задатака (2/4)

- Пример: матрица задатака MxN где сваки задатак зависи од задатка лево и изнад себе:



  - Сваки задатак рачуна збир улаза са леве и горње стране.

Општи ациклични граф задатака (3/4)

```
const int M=3, N=4;

class DagTask: public tbb::task {
public:
    const int i, j;
    // input[0] = sum from above, input[1] = sum from left
    double input[2];
    double sum;
    // successor[0] = successor below, successor[1] = successor to right
    DagTask* successor[2];
    DagTask( int i_, int j_ ) : i(i_), j(j_) {
        input[0] = input[1] = 0;
    }
    task* execute() {
        __TBB_ASSERT( ref_count()==0, NULL );
        sum = i==0 && j==0 ? 1 : input[0]+input[1];
        for( int k=0; k<2; ++k )
            if( DagTask* t = successor[k] ) {
                t->input[k] = sum;
                if( t->decrement_ref_count()==0 )
                    spawn( *t );
            }
        return NULL;
    }
};
```

Општи ациклични граф задатака (4/4)

```
double BuildAndEvaluateDAG() {
    DagTask* x[M][N];
    for( int i=M; --i>=0; )
        for( int j=N; --j>=0; ) {
            x[i][j] = new( tbb::task::allocate_root() ) DagTask(i,j);
            x[i][j]->successor[0] = i+1<M ? x[i+1][j] : NULL;
            x[i][j]->successor[1] = j+1<N ? x[i][j+1] : NULL;
            x[i][j]->set_ref_count((i>0)+(j>0));
        }
    // Add extra reference to last task, because it is waited on
    // by spawn_and_wait_for_all.
    x[M-1][N-1]->increment_ref_count();
    // Wait for all but last task to complete.
    x[M-1][N-1]->spawn_and_wait_for_all(*x[0][0]);
    // Last task is not executed implicitly, so execute it explicitly.
    x[M-1][N-1]->execute();
    double result = x[M-1][N-1]->sum;
    // Destroy last task.
    task::destroy(*x[M-1][N-1]);
    return result;
}
```

Садржај

- Увод
- Програмирање засновано на задацима
- Када га не користити?
- Пример
- Како ради распоређивач?
- Празан задатак
- Општи ациклични граф задатака
- Преглед

Преглед (1/2)

- Распоређивач задатака ефикасно ради код паралелизма са гранањем и придруживањем (*fork-join*), где има доста гранања, тако да преузимање задатака може да омогући довољну ширину у извршавању задатака и тиме заузима нити, које извршавају задатке по дубини, док могу.
- Распоређивач задатака није једноставан јер је створен за брзину.
- Ако мора директно да се користи, најбоље је сакрити га испод виших нивоа, попут шаблона `parallel_for`, `parallel_reduce` i sl.

Преглед (2/2)

- Важно:
  - Увек треба користити `new(`*`metoda dodele`*`)` за доделу задатка, где је *`metoda dodele`* једна од метода класе `task`.
  - Сви задаци на истом нивоу морају бити додељени пре покретања, осим ако се користи `allocate_additional_child_of`.
  - Ако се задатак заврши, и ако није означен за поновно извршавање, аутоматски се уништава. Број веза његовог наследника се умањује и ако је достигао 0, наследник се аутоматски покреће.