Programování „bez vláken“

- „Tradiční“ přístup je vytvoření vícevláknového programu, kde se řekne, co má které vlákno dělat
- Ale jde to i jinak, lze vytvořit program tak, že se řekne, co se má udělat paralelně
- OpenMP, Intel Thread Building Blocks, MS Concurrency Runtime

OpenMP

- Open Multi-Processing
- Idea je, že se vezme sériový program a za použití speciálních knihoven se pomocí direktiv překladače řekne, co se má provádět paralelně
- Např. pokud se překladač vyvolá s direktivou openmp, program se přeloží jako paralelní, bez ní jako sériový

```
#pragma omp parallel
{
  for (int i=0; i<10; i++) {
          DoSomething();
    }
}
```

- Pokud se provádí paralelní výpočet, obvykle je nutná nějaká sdílená/redukční/uzamykaná proměnná
- Překladač se je může pokusit uhodnout, ale to se vždy nemusí podařit správně
    - Je lepší mu to prozradit pomocí speciálních direktiv
    - shared, private, reduction, default…
- To samé platí i pro synchronizaci a plánování

- Lze použít i direktivu if, která spustí výpočet paralelně pouze tehdy, je-li daná podmínka splněna
  - Např. pokud je počet prvků pole k součtu dostatečně velký, aby se vyplatilo sečíst je paralelně

- Výhodou je, že se dá „doparalelizovat“ sekvenční program, aniž by se do něho muselo nějak výrazně zasahovat
  - Např. výpočet statistiky

```
//#pragma omp parallel for
// reduction (+:localAvgDiff, localAvgAbsDiff)
// private(diff) shared(localMaxAbsDiff)

#pragma omp parallel
{
   floattype sublocalMaxAbsDiff = 0.0;
  //so that we do not need to declare localMaxAbsDiff
  //as shared, thus intruducing wait penalty

  #pragma omp for nowait
          reduction (+:localAvgDiff, localAvgAbsDiff)
          private(diff)
  for (i=0; i<cnt; i++) {
    diff = leftBuf[i].y-rightBuf[i].y;

    localAvgDiff += diff;

    diff = fabs(diff);
    localAvgAbsDiff += diff;

#ifdef _MSC_VER
     sublocalMaxAbsDiff = max(sublocalMaxAbsDiff,
                              diff);
#else
     sublocalMaxAbsDiff = fmax(sublocalMaxAbsDiff,
                               diff);
#endif
   } //for
  #pragma omp critical (localMaxAbsDiff)
```

```
  localMaxAbsDiff = max(localMaxAbsDiff,
                        sublocalMaxAbsDiff);
} //pragma omp parallel


if (cnt) {
  //if to avoid division by zero
  floattype tmp;
  tmp = valuescnt;
  tmp = 1.0/tmp;

  (*stats).AvgAbsDiff = localAvgAbsDiff*tmp;
  (*stats).AvgDiff = localAvgDiff*tmp;
  (*stats).MaxAbsDiff = localMaxAbsDiff;
}
```

Intel Thread Building Blocks

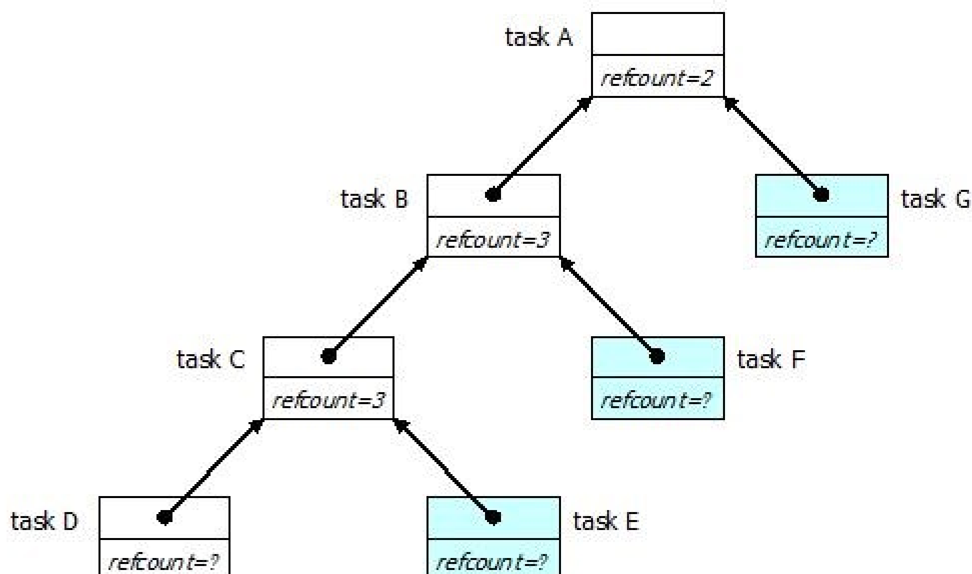
- Open Source, podpora více platforem
  - Stejně jako OpenMP
- Ačkoliv si je TBB vědoma vláken poskytovaných OS, myšlenka je takový, že programátor už s vlákny nepracuje
- Namísto vláken specifikuje úlohy, tasks, a jejich návaznosti
- Důvodem je mj. redukce efektu cache-cooling

  - Uvažujeme tradičních m vláken na n procesorů, kde m > n
  - Jak vlákno běží, jeho pracovní data se dostávají do cache procesoru (tzv. hot data)
  - Jak se vlákna přepínají, do cache procesoru se dostávají pracovní data nového vlákna a pracovní data předchozích vláken jsou z cache odstraňována
  - Takže až takové vlákna přijde opět na řadu, každý cache-miss ho bude stát stovky cyklů čekání, než se jeho pracovní data znovu nahrají do cache (protože byly tzv. cooled)

  - S TBB programátor nepíše vlákna, ale úlohy
  - Vlákna poskytuje TBB tak, aby m<=n
  - Vlákno TBB pracuje na jedné úloze, dokud není dokončena => redukuje efekt cache-cooling

  - Tradiční vlákna ani thread-pool nic takového neumí!
    - =>tradiční vlákna jsou dobrá na GUI a I/O, ale ne na výpočty

- Task Stealing je způsob plánování úloh



http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/tbb_userguide/How_Task_Scheduling_Works.htm

- Úlohy A, B, C vytvořily podúlohy, na které čekají
- Úloha D běží
- Úlohy E, F, G ještě neběží

- Nejhlouběji zanořené úlohy jsou nejvíce cache-hot
- Breadth-first co nejdříve rozbalí celý strom, takže maximalizuje paralelismus
- Depth-first zase udržuje omezený počet uzlů stromu
- Takže je to v praxi kompromis, protože počet procesorových jader je omezený

- Na každý procesor je zásobník úloh
- Pokud jeden procesor dokončí svůj zásobník, může zkusit „ukrást“ cool data z vrcholu zásobníku jiného procesoru

- Aneb FIFO ani priority nejsou vždy ta nejlepší strategie

- TBB vykonává úlohy paralelně, jak nejlépe to se současným know-how jde
    - Obsahuje různé další optimalizace v jakém pořadí úlohy vykonávat
    - Lze si napsat vlastní plánovač

- Stejně jako STL, i TBB intenzivně používá C++ šablony
    - Tj. nelze ji použít v C jako OpenMP
- Základní konstrukce TBB jsou
    - parallel_for, parallel_reduce, parallel_scan
    - parallel_while, parallel_do,pipeline, parallel_sort
    - paralelní kontainery, které jsou v STL
    - mutex, spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex, recursive mutex
    - fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store

- TBB dokáže použít vlastní paměťový manažer, který je optimalizovaný na paradigma výpočtu úloh

- TBB je náročnější se naučit, ale zase se to vyplatí na výkonu, pokud se začíná psát nová aplikace, než např. paralelizovat s OpenMP
    - Navíc TBB se nespoléhá na direktivy pro překladač, takže podmínka if OpenMP se dá realizovat libovolně složitá, nebo se dají udělat konstrukce, který by šli s OpenMP udělat jen velmi obtížně – např. cancellation výpočtu v OpenMP není

- Např. chceme-li spustit na pozadí několik výpočetně náročných úloh paralelně

```
tbb::task* CMasterCalculationTBBLogic::execute() {
 tbb::task_list list;

 for (int i=gaiFirst; i<=gaiLast; i++)
  list.push_back(*new(allocate_child()) CTask(i));

 set_ref_count(gaiLast-gaiFirst+2); //počet úloh +1
 spawn_and_wait_for_all(list);

 return NULL; //non-NULL by byla úloha, která by měla
}         //být spuštěna jako další/závislá na téhle
```

- Když bychom např. násobili vektor

```
class CMulVect {
    floattype *mA, *mB;
    int mLen;
public:
    floattype mProduct;

    CMulVect(floattype *a, floattype *b, int len) :
            mA(a), mB(b), mLen(len), mProduct(0.0) {}
    CMulVect(CMulVect& x, tbb::split) : mA(x.mA),
             mB(x.mB), mLen(x.mLen), mProduct(0.0) {}

    void join(const CMulVect& y) {
        mProduct += y.mProduct;
    }

    void operator()( const
                tbb::blocked_range<size_t>& r ) {
        int r_end = r.end();

        //Taken from Intel's tutorial on TBB
        //by declaring these variables, we make it
         //obvious to the compiler that they
        //can be held in registers instead in memory
```

```
         // => speedup

         floattype *a = mA;
         floattype *b = mB;
         floattype sum = 0.0;

         for (int i=r.begin(); i!=r_end; ++i)
             sum += a[i]*b[i];

         mProduct += sum;
    }
};

floattype MulVect(floattype *a, floattype *b,
                  int len) {

    floattype result = 0.0;

    //Jsou data tak velká, aby se je vůbec
    //vyplatilo počítat paralelně?
    if (len<=rmSerialThreshold) {
       for (int i=0; i<len; i++)
          result += a[i]*b[i];
    } else {
        CMulVect mul(a, b, len);

        tbb::parallel_reduce(
          tbb::blocked_range<size_t>(0,len), mul);
        result = mul.mProduct;
    }
    return result;
}
```

- U redukčních operací je třeba při inicializaci mezivýsledku rozlišovat mezi konstruktorem a funkčním operátorem ()
    - Konstruktory jsou dva
        - Normální, který bere parametry výpočtu
        - A split konstruktor, který bere referenci na druhý objekt a na tbb::split

  - V konstruktorech se vždy inicializují dílčí mezivýsledky celé operace
  - Návrhově to totiž může svádět k tomu, aby se mezivýsledek inicializoval v těle funkčního operátoru
  - Jenže funkční operátor se může volat několikrát na stejné instanci, a pak by to dávalo špatné výsledky

Případová studie použití TBB

- Uvažujme výpočetně náročnou aplikaci nad rozsáhlými daty s GUI
- Jedno vlákno bude obsluhovat GUI
- Jedno vlákno bude obsluhovat I/O
  - Naivní přístup je jedno vlákno pro zápis a jedno pro čtení
  - Lepší je jenom jedno vlákno a použití asynchronních I/O operací
    - OS si je uspořádá sám pro lepší výkon
    - Např. disky mají Native Command Queuing
  - Ale co s výpočetní částí?
    - Určitě v tom bude alespoň jedno vlákno, aby výpočet běžel na pozadí a GUI bylo responsive
- Přístup s psaním vláken by znamenal postarat se
  - Vytváření a rušení vláken, což je další režie pro OS
  - Jejich správnou synchronizaci, což je náročné, jakmile se program stává složitější
  - Výkonnostně je rozdíl, jestli se synchronizuje v kernel-mode, nebo v user-mode address space
  - Výkonnostně hraje roli, zda se k datům přistupuje ze stejného procesoru – thread affinity

    - Optimální počet vláken odpovídá počtu procesorových jader v systému, což ale není přístup, který je vždy možný při psaní vláken
        - Např. počet vláken může odpovídat tomu, jaká je metoda výpočtu – problém škálovatelnosti
            - S TBB se taková vlákna nahradí úlohami

- S TBB
    - O výše uvedené problémy se TBB postará
    - Programátor napíše jedno vlákno, ve kterém pak spustí výpočet úloh TBB
    - Nicméně programátor si stále musí být vědom toho, jak se píší paralelní programy s vlákny, aby mohl správně používat synchronizační primitiva TBB
        - Optimálně se naprogramují pouze úlohy s tím, že každá úloha po dokončení vrátí další úlohu, která se má vykonat jako navazující
        - Jako bariéra na dokončení více úloh se pak použije task list
        - Úlohy však mohou sdílet proměnné, a proto je třeba znát teorii o psaní vláken
            - Aby byla představa, že 2 úlohy mohou, ale i nemusí, běžet paralelně, např. když se má správně použít mutex, podmínka...

Concurrency Runtime

- Proprietární technologie MS
- Ideově podobná TBB, je to novější knihovna
- Jenže TBB běží nejen pod Windows a výkonově nezaostává

- Spuštění více úloh

```
void Task1() {}
void Task2() {}


void Main() {
    task_group tg;

   tg.run(&Task1);
   tg.run(&Task2);

   tg.wait();
}
```

- Výpočet součtu sum
  - http://msdn.microsoft.com/en-us/magazine/ee309514.aspx

```
combinable<int> localSums;

parallel_for_each(myVec.begin(),  myVec.end(),
                  [&localSums] (int element) {
   SomeFineGrainComputation(element);
   localSums.local() += element;
});
```

- Násobení matic
    - http://msdn.microsoft.com/en-us/library/dd728073.aspx

```
void parallel_matrix_multiply(double** m1,
double** m2, double** result, size_t size)
{
   parallel_for (size_t(0), size, [&](size_t i)
   {
      for (size_t j = 0; j < size; j++)
      {
         double temp = 0;
         for (int k = 0; k < size; k++)
         {
            temp += m1[i][k] * m2[k][j];
         }
         result[i][j] = temp;
      }
   });
}
```