Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

DIPLOMOVÁ PRÁCE

Plzeň, 2002

Josef Kohout

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Paralelní Delaunayova triangulace ve 2D a 3D

Plzeň, 2002 Josef Kohout

Parallel Delaunay triangulation in 2D and 3D

The construction of the Delaunay triangulation belongs to the fundamental problems in computer graphics, that's why many different parallel algorithms exist. This thesis describes the most known parallel solutions and suggests a new parallel algorithm that is based on the well-known sequential algorithm for construction of the Delaunay triangulation by randomized incremental insertion. This algorithm is very popular due to its simplicity and robustness. The developed algorithm works with one shared data structure, so there is no need to merge sub-results into the resulting mesh; and our algorithm seems to be simplest. It was tested on multiprocessors with 2 to 8 processors using up to 8 threads and the achieved speed-up is similar to the results of other parallel algorithms. The newly proposed algorithm is usable in 2D and also in 3D and is intended for large community of users of the multiprocessor architectures with shared memory and only several processors.

Obsah

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne .....

.....

Úvod

1 Nástin problematiky

1.1 Problém triangulace

Obor počítačová grafika a virtuální realita je založen na modelování reálného světa. Úlohy počítačové grafiky simulují jeho chování, zobrazují vhodným způsobem jeho stav nebo konkrétní náhled na něj. Většina těchto úloh spočívá ve zpracování často velkého množství dat, které byly naměřeny ve zvolených místech reálného systému. V jednorozměrných případech bývají těmito místy většinou časové okamžiky [*t*], ve dvourozměrných případech to bývají body v rovině, nejčastěji o souřadnicích [*x*,*y*], a ve trojrozměrných případech pak body v prostoru o souřadnicích [*x*,*y*,*z*]. V této práci budeme uvažovat pouze 2D a 3D data.

Ve zvolených bodech se měří jak skalární hodnoty, jako je např. teplota, tlak, rychlost proudění částice, výška; tak i vektorové hodnoty, jako je např. směr proudění v daném místě.

Úkolem je vhodným způsobem prezentovat člověku tyto diskrétní hodnoty stanovené v diskrétních bodech. Pouze v nejjednodušším případě postačuje prezentovat hodnoty jen naměřených bodů. Většinou je nezbytné odvodit hodnoty i v bodech, kde měření neproběhlo. Hodnotu lze odvodit interpolací hodnot z okolních bodů. Je zřejmé, že bližší bod má větší vliv na výslednou hodnotu. Otázkou zůstává, co jsou okolní body? Jsou-li body uspořádány do pravidelných mřížek, je situace jednoduchá. Co si však počít, když body nejsou nijak organizovány a jsou rozptýlené?

Často se proto přidává k množině bodů *S* ještě množina neprotínajících se úseček *L* s krajními body z množiny *S* [Mag98]. Tyto úsečky tvoří mnohoúhelníky (resp. mnohostěny v trojrozměrném případě). Pro pravidelná data se nejčastěji používají čtverce, pro obecnou organizaci vstupních dat je nejjednodušším a nejčastějším primitivem trojúhelník (resp. tetrahedron). Vzniklá struktura {*S*, *R*}, kde *R* je množina polygonů, se nazývá sítí, v případě trojúhelníků mluvíme o trojúhelníkové síti nebo také o triangulaci. Obdobně ve 3D se jedná o síť tetrahedronů nebo také tetrahedronizaci. V literatuře se však často také používá pojem triangulace i pro trojrozměrný případ.

Výhoda reprezentace {*S*, *R*} spočívá v tom, že pro vnitřní body polygonu jsou sousedními body vrcholy polygonu. Interpolace v případě známého seznamu polygonů je velmi jednoduchá, navíc interpolace bodů na trojúhelníku je podporována hardwarově (problematika stínování). Nadále se budeme zabývat pouze problémem, jak zkonstruovat množinu *R* na základě množiny vstupních bodů.

1.2 Hlavní cíle

Proces konstrukce trojúhelníkové a tetrahedronové sítě se nazývá jednoduše triangulací. Speciálně pro proces konstrukce sítě tetrahedronů uvádí literatura pojem tetrahedronizace.

Protože jde o důležitý problém, existuje velké množství efektivních metod. Jednou z nejdůležitějších a nejznámějších je Delaunayova triangulace. Doba potřebná pro výpočet však i přes rychlost moderních počítačů nemusí být zcela dostačující (zejména pro větší množiny dat). Paralelizace patří mezi prostředky, jak redukovat čas potřebný pro výpočet.

V dřívějších dobách, kdy k paralelnímu výpočtu neodmyslitelně patřily nákladné superpočítače, vzniklo několik paralelních algoritmů pro konstrukci Delaunayovy triangulace. Tyto algoritmy však často vyžadují speciální paralelní architektury a/nebo jsou značně složité, a proto nejsou v dnešní době zcela ideální. V současnosti díky nízké cenové hladině výpočetní

techniky totiž stále přibývá uživatelů vlastnících víceprocesorový počítač se sdílenou pamětí a několika (většinou dvěma) procesory. Tito uživatelé nepotřebují algoritmy, které dosahují špičkového urychlení pro tisíce procesorů, ani algoritmy, jejichž implementace si vyžádá značného úsilí.

Cílem této práce je paralelizovat algoritmus konstrukce Delaunayovy triangulace metodou inkrementálního vkládání. Předností tohoto algoritmu je jeho jednoduchost a stabilita, ale díky svému ryze sekvenčnímu charakteru se jakákoliv paralelizace algoritmu nezdá být vhodná, uvažujeme-li o nasazení vyššího počtu výpočetních jednotek Výsledný paralelní algoritmus je proto „ušit na míru“ uživatelům, kteří nevlastní superpočítač o tisících procesorech, ale již zmiňované počítače s nejvýše desítkami procesorů.

1.3 Poděkování

Poděkování patří všem, kteří se (byť nevědomky) podíleli na tomto projektu. Největší díky zasluhuje vedoucí této práce, Doc. Dr. Ing. Ivana Kolingerová, která kromě teoretického zázemí poskytla rovněž odladěnou sériovou verzi algoritmů *DT* (jak 2D tak 3D) ve zdrojové formě Borland Delphi[1].

Ověření navržených paralelních algoritmů na víceprocesorových počítačích s více než dvěma procesory opakovaně umožnila společnost Dell Computer[2], Czech Republic. Určitou pozornost si zasluhuje rovněž osmi procesorový stroj ES5000 společnosti UniSys Praha, na němž proběhl jeden z testů.

Diplomová práce byla řešena v rámci projektů Ministerstva školství, mládeže a tělovýchovy České Republiky MSM 23500005 a AV2030801.

Společnost Microsoft si zasluhuje díky za velmi kvalitní vývojové nástroje Microsoft Visual C++ 6.0 a Microsoft Visual Studio.NET 7.0 Beta 2[3], které byly použity při implementaci navržených paralelních algoritmů (jak samostatné verze, tak verze pro MVE[4]).

Odladěný algoritmus pro výpočet mediánu byl vytvořen kolegy Šimánem a Krocem.

1.4 Používané zkratky

Zkratky, které nejsou uvedeny v této podkapitole, jsou buď všeobecně používané (např. atd., apod., aj., tj.), anebo mají pouze vztah k podkapitole (resp. ke kapitole), ve které jsou použity, a tam jsou také vysvětleny.

2D	E^2	Dvourozměrný (rovinný) případ
3D	E^3	Trojrozměrný (prostorový) případ
CH(S)		Konvexní obálka
D&C	Divide & Conquer	Rozděl a panuj
DAG	Directed Acyclic Graph	Orientovaný acyklický graf
DT	DT(S)	Delaunayova triangulace množiny bodů S
MS		Microsoft
N		Počet bodů vstupní množiny
PDT		Paralelní Delaunayova Triangulace
PEs	Processing Elements	Počet procesorů

[1] Borland Delphi je produktem Inprise Corporation (http://www.borland.com)

[2] Dell PowerEdge 7150 a Dell PowerEdge 8450 jsou počítače společnosti Dell Computer (http://www.dell.com)

[3] Oba nástroje jsou produktem Microsoft Corporation (http://www.microsoft.com)

[4] MVE – Modular Visualisation Environment je vizualizační nástroj vyvinutý na Západočeské Univerzitě.

1.5 Základní pojmy a terminologie

V této kapitole jsou uvedeny základní pojmy, které jsou v práci použity.

Konvexní obálka množiny bodů je nejmenší konvexní množina, která obsahuje všechny zadané body uvnitř, resp. ve vrcholech obálky. Označuje se *CH(S)*, kde *S* je množina bodů.

Asymptotická složitost algoritmu vypovídá o rychlosti růstu počtu časových jednotek spotřebovaných algoritmem v závislosti na počtu vstupních jednotek *N*, kde *N* je větší než konstanta N_0 (pro triangulace je *N* počet vstupních bodů).

Každý přístup do paměti i provedení každé jednoduché operace (+,-,*,/,=,if,call) vyžaduje jednu časovou jednotku. Uvažujme všechna možná vstupní data pro konkrétní počet vstupních jednotek *N*, potom lze stanovit maximální, minimální a průměrný počet časových jednotek potřebných algoritmem pro zpracování libovolné datové množiny s tímto konkrétním *N*. Funkci $f_{max}(N)$, která přiřazuje každému *N* odpovídající maximální počet časových jednotek, nazveme složitostí algoritmu pro nejhorší případ. Analogicky funkci $f_{min}(N)$ přiřazující odpovídající minimální počet časových jednotek každému *N* nazveme složitostí algoritmu pro nejlepší případ a funkci $f_{avg}(N)$ nazveme složitostí algoritmu pro průměrný případ.

Pro hodnocení algoritmu se používají asymptotické verze těchto funkcí, nutno však poznamenat, že platí pouze pro dostatečně velká *N*. Nechť existují konstanty c_1, c_2 a $N_0 > 0$ a složitosti *f(N)* a *g(N)*. Pokud platí, že $c_1 \cdot g(N) \leq f(N) \leq c_2 \cdot g(N)$, potom funkce *f* a *g* popisují stejnou algoritmickou složitost, což zapíšeme: $f(N) = \Theta(g(N))$. Například složitost algoritmu pro násobení dvou *NxN* matic přímou aplikací strategie D&C je: $f(N) = 8 \cdot f(N/2) + 4 \cdot \Theta(1)$, což je $\Theta(N^3)$. Často je obtížné nalézt vhodnou funkci *g(N)* takovou, že funkční hodnoty *f(N)* budou omezeny jak zdola tak shora. Předpokládejme existenci konstanty *c* a konstanty N_0, potom zápis $\Omega(g(N))$ znamená, že pro $N > N_0$ je *f(N)* větší rovno $c \cdot g(N)$, a zápis $O(g(N))$ značí, že pro $N > N_0$ je *f*(*N*) je menší rovno $c \cdot g(N)$, tzv. horní mez výkonnosti algoritmu.

$O(N)$ tj. lineární závislost znamená, že čas potřebný pro výpočet je přímo úměrný velikosti vstupních dat. Oproti tomu pro algoritmus s kvadratickou složitostí $O(N^2)$ platí, že zdvojnásobí-li se velikost vstupních dat, čas vzroste 4x. Málokterá data jsou tak špatná, aby nastal nejhorší případ, proto se uvádí často také tzv. očekávaná složitost. Ta bývá většinou stanovena teprve na základě implementace algoritmu, protože analýza není jednoduchá.

Distribuce má český význam rozdělení. V této práci se používá ve dvou významech, a to pro vyjádření operace rozdělení práce mezi jednotlivé výpočetní jednotky, a vyjádření uspořádání vstupních bodů (např. uniformní rozdělení).

Singularita je termín často používaný ve spojení s charakterem vstupních dat. Singulární případem jsou vstupní data se speciálním charakterem, která však nejsou v množině různých dat příliš častá, spíše ojedinělá.

Simplex je útvar, který se nachází v dimenzi *D* a současně má *D+1* vrcholů. V jednorozměrném případě se jedná o úsečku, ve dvourozměrném pak o trojúhelník a v prostoru se jedná o čtyřstěn (tetrahedron).

Program je statický zápis výpočetního postupu, nemá stav a nemění se.

Proces je aktivita probíhající v čase podle programu. Stav této aktivity je určen aktuálním místem v programu a okamžitými hodnotami zpracovávaných dat.

Vlákno se z hlediska paralelního programování neliší od procesu. Odlišnost lze vyzkoumat z hlediska efektivity, vlákno vyžaduje zpravidla nižší režii. V této práci budeme tento termín používat jako synonymum termínu proces.

1.6 Organizace textu

Text celé práce je rozdělen do tří základních částí, a to na teoretickou část, realizační část a závěr. Teoretická část nejprve definuje Delaunayovu triangulaci, popisuje její vlastnosti a různé přístupy její konstrukce. Přístupu konstrukce *DT* metodou inkrementálním vkládáním je věnována další samostatná kapitola. Dále je k popsaným přístupům konstrukce *DT* uvedeno několik nejznámějších paralelních algoritmů.

V realizační části je, na základě analýzy existujícího sekvenčního algoritmu, navrženo několik způsobů paralelizace, které byly implementovány, a komentované praktické výsledky, včetně srovnání s popsanými existujícími paralelními algoritmy Delaunayovy triangulace jak ve 2D, tak i ve 3D, jsou uvedeny ve formě grafů a tabulek v samostatné kapitole následující po stručném popisu systému MVE.

Závěrečná část shrnuje celou tuto práci (včetně dosažených praktických výsledků) . Po závěrečné části následují tři přílohy označené písmeny A, B a C. Příloha A obsahuje obrázky, grafy, tabulky či algoritmy, které nepatří mezi nezbytné objekty náležející přímo do textu práce, a navíc jsou příliš rozsáhlé. Na všechny objekty umístěné v příloze A jsou odkazy v textu práce. Příloha B popisuje programové vybavení aplikace a uživatelské rozhraní. Na tuto přílohu navazuje příloha C uvádějící začlenění řešení do MVE a příklady použití implementovaných modulů v tomto prostředí.

Teoretická část

2 Delaunayova triangulace

Delaunayova triangulace *DT(S)* definovaná nad množinou vstupních bodů *S* je množina trojúhelníků takových, že

1. bod $p \in \mathrm{E}^2$ je vrcholem trojúhelníku tehdy a jen tehdy, pokud patří do množiny *S*.
2. průsečík dvou trojúhelníků z množiny *DT(S)* je buď prázdný, anebo se jedná o společnou hranu nebo vrchol.
3. kružnice opsaná[5] každému trojúhelníku neobsahuje žádný další bod z množiny *S*.

Definice Delaunayovy triangulace *DT(S)* ve 3D je analogická k výše uvedené definici. *DT(S)* pak je množina čtyřstěnů takových, že

1. bod $p \in \mathrm{E}^3$ je vrcholem čtyřstěnu tehdy a jen tehdy, pokud patří do množiny *S*
2. průsečík dvou čtyřstěnů z množiny *DT(S)* je buď prázdný, anebo se jedná o společnou stěnu, hranu nebo vrchol.
3. koule opsaná[6] každému čtyřstěnu neobsahuje žádný další bod z množiny *S*.

[Kol99] uvádí také alternativní definici Delaunayovy triangulace, podle které je *DT* duálem Voronoiova diagramu *Vor(S)*. Voronoiův diagram je množina všech bodů, které mají stejnou vzdálenost od více než jednoho bodu z množiny *S* a současně neexistuje žádný další bod, jehož vzdálenost by byla menší. Matematicky lze *Vor(S)* zapsat:

$$Vor(S) = \{x \in E^d : \forall p_k, \exists p_i, p_j; p_i, p_j, p_k \in S; i \neq j \neq k \neq i : |p_k - x| \geq |p_i - x| = |p_j - x|\}$$

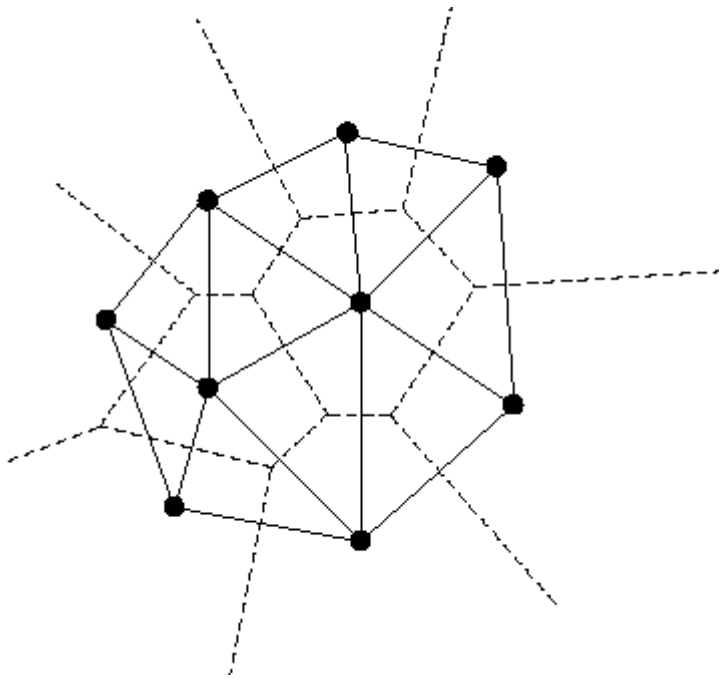
Vzájemný vztah *Vor(S)* a *DT(S)* je uveden na obrázku 2.1. Existují algoritmy, které řeší konstrukci *DT* přes Voronoiův diagram, ale jsou méně efektivní než přímé metody.

Podstata jednotlivých přímých metod pro konstrukci Delaunayovy triangulaci bude uvedena v další podkapitole (převážně podle [Cig93, Mag98, Kol99, Ble96 a Su94]). Zde uveďme jen výčet přímých metod:

- lokální zlepšování
- inkrementální vkládání (alias on-line)
- inkrementální konstrukce
- zametání
- převod do vyšší dimenzi
- rozděl a panuj (D&C)

[5] V anglických originálech se používá termín circle, který v češtině má význam kružnice, kruh. Výrok kruh opsaný však není matematicky správný, výhradně se užívá pojem kružnice opsaná. Při takovém překladu do českého jazyka však zbytek definice není korektní; uvedené pravidlo v definici budeme proto chápat tak, že žádný další bod z množiny neleží na hraně ani uvnitř kruhu se stejným středem a poloměrem jako kružnice opsaná tomuto trojúhelníku.

[6] Pod pojmem koule opsaná rozumějme kulovou plochu opsanou nebo lépe kouli sestrojenou tak, že všechny vrcholy simplexu leží na jejím povrchu.



Obr. 2.1: *Delaunayova triangulace (plné linie) a Voronoiův diagram (čárkované linie) množiny bodů ve 2D [Mag98].*

Vzhledem ke svým vlastnostem je Delaunayova triangulace velmi oblíbená. V oblibě je zejména dvourozměrná varianta *DT*, *3D DT* nemá již příliš mnoho výhodných vlastností. *2D DT* podle [Ber97, citováno dle Kol99]

1. minimalizuje poloměr kružnice opsané a maximalizuje minimální úhel (ačkoliv neminimalizuje maximální úhel), tj. negeneruje příliš hubené trojúhelníky, které mohou být problematické při zobrazování a dalším zpracování.
2. Pokud žádné čtyři body neleží na kružnici, je *DT* jednoznačná.
3. Hranice *DT(S)* je *CH(S)*.

Poslední uvedená vlastnost nemusí být vždy žádoucí, v některých případech (např. při rozpoznávání) je vyžadována nekonvexnost či dokonce přítomnost některých hran. Existuje modifikace tzv. „Constrained Delaunay Triangulation“ (*CDT*), která tento problém řeší.

Delaunayova triangulace a zejména její *CDT* varianta má široké uplatnění [Har97, Cig92]. Je používána nejen pro modelování terénů, ale také v oblasti rozpoznávání. Ať již se jedná o rozpoznávání tvarů [Pra00] nebo např. spektrální analýzu v krystalografii či analýzu EKG apod. Možné použití je v meteorologii, robotice, výpočtové geometrii, chemii i biologii. V oblasti počítačové grafiky je *DT* používána pro stochastický dithering [Ost99], volumetrický rendering nebo 3D verze také pro rekonstrukci povrchu z bodů v prostoru.

2.1 Lokální zlepšování

Metoda *lokálního zlepšování* je použitelná výhradně pro konstrukci dvourozměrné varianty Delaunayovy triangulace. Nejprve je sestrojena libovolná triangulace *T*, která je ve druhém kroku postupně převáděna na *DT*.

Delaunayova triangulace je taková triangulace *T*, ve které všechny hrany jsou lokálně optimální. Hrana e libovolné triangulace *T* je lokálně optimální tehdy a jen tehdy, pokud čtyřúhelník *Q*, který je tvořen dvěma sousedními trojúhelníky se společnou hranou *e*, je buď ne-

konvexní, anebo nahrazení diagonály *e* druhou diagonálou *e'* by nezvětšilo minimální úhel šesti vnitřních úhlů vzniklé triangulace v *Q*. Pokud hrana není lokálně optimální, je nutné provést záměnu těchto hran *e* a *e'*.

Ačkoliv implementace je jednoduchá, stabilita algoritmů velmi dobrá, v praxi se tyto metody téměř nepoužívají. Složitost závisí na způsobu konstrukce primární triangulace, obvykle bývá v nejhorším případě $O(N^2)$ a $O(N)$ pro průměrný případ.

2.2 Inkrementální vkládání

Metoda *inkrementálního vkládání* je stěžejní metodou navrhovaného řešení, a proto bude detailně popsána v samostatné kapitole. V této podkapitole uveďme pouze, pro srovnání s ostatními metodami, podstatu inkrementálního vkládání.

Implementačně se jedná o nejjednodušší metodu, ale její časová složitost v nejhorším případě bývá $O(N^2)$, protože nově vytvářené hrany triangulace mohou být otestovány vůči všem již přijatým hranám. Randomizací množiny bodů *S* však lze docílit celkového času algoritmu $O(N \cdot \log N)$. Body jsou po jednom postupně přidávány do již existující *DT*, což umožňuje, aby při zahájení algoritmu nebylo nutné znát všechny vstupní body, pouze interval jejich souřadnic. Proto je tato metoda označovány rovněž jako on-line.

Po vložení bodu vznikne nová triangulace, která však nemusí splňovat Delaunayův princip prázdné kružnice opsané (koule ve 3D). Proto je nutné provést opravu nově vzniklé triangulace. Ta spočívá stejně jako u metod lokálního vylepšování v prohození hran (stěn).

Po prohození hrany však je opět nezbytně nutné ověřit, zda se i nadále jedná o *DT*. Vložení jednoho bodu proto v nejhorším případě může vyvolat změnu celé sítě. Svou povahou se proto jedná o sériový algoritmus.

2.3 Inkrementální konstrukce

Ke konstrukci *DT* metodou *inkrementální konstrukce* je využíváno vlastnosti prázdné kružnice opsané (koule ve 3D). Pro jednoduchost výkladu, popíšeme pouze podstatu metody pro dvourozměrný případ, trojrozměrný případ se příliš neliší.

Konstrukce začíná volbou libovolného počátečního bodu, vyhledáním od něj nejbližšího bodu a sestrojením první hrany. Postupně je pro každou vnější hranu *DT* vyhledán v množině vstupních bodů takový, pro který je poloměr kružnice opsané nejmenší. Tento bod spolu s hranou tvoří nový trojúhelník (a tudíž i nové vnější hrany triangulace).

Metoda inkrementální konstrukce je jednoduchá, ale pokud algoritmus neobsahuje účinné urychlovací techniky pro vyhledávání, má nízkou efektivitu (pro nejhorší případ je složitost trojrozměrné verze $O(N^3)$).

2.4 Zametání

Metoda *zametání* je metodou určenou především pro dvourozměrný případ (rozšíření pro 3D sice existuje, ale je natolik komplikované, že se v praxi téměř nepoužívá). Tato metoda pohybující se horizontální přímkou postupně „smete" všechny vstupní body. Přímka rozděluje rovinu na oblast, která již byla zpracována (v té se nachází dílčí triangulace), a na oblast, která teprve musí být zpracována (v té se nachází body, které se postupně přidají k již existující triangulaci – principielně velmi obdobné inkrementální konstrukci). Dorazí-li přímka do nového bodu, je sestrojena potencionální hrana mezi tímto a předchozím bodem. Tato hrana je vložena do seznamu zvaného *frontier*. Jakmile je vložena druhá hrana, lze sestrojit kružnici opsanou posledním třem bodům. Nenalezne-li algoritmus v dalších krocích v této kružnici žádný další bod, tvoří tyto tři body trojúhelník ve výsledné triangulaci.

Jedná se, dle mého mínění, o relativně komplikovanou metodu, nicméně její složitost pro nejhorší případ je *O(N· log N)*. Podrobnosti o této metodě lze nalézt v [For87, Su94].

2.5 Převod do vyšší dimenze

Převod do vyšší dimenze je poněkud komplikovanějším způsobem konstrukce *DT* (opět se jedná o metodu, která se v praxi pro konstrukci *3D DT* téměř nepoužívá).

Je dokázáno, že úlohu konstrukce *DT* lze převést na problém konstrukce konvexní obálky v dimenzi o jednu vyšší než je dimenze úlohy původní. Vstupní body z E^d jsou transformovány do E^{d+1}, je spočítána konvexní obálka a zpětnou projekcí konvexní obálky se obdrží požadovaná *DT*.

2.6 Rozděl a panuj (D&C)

Rozděl a panuj (D&C) není plnohodnotná metoda (ve srovnání s předchozími). Je založena na rekurzivním dělení vstupní množiny bodů, dílčí triangulaci dané podmnožiny a v poslední fázi na sloučení výsledných triangulací. Dílčí triangulace je provedena algoritmem kategorizovaným do některé z již uvedených tříd.

Implementace D&C algoritmů je sice obtížnější, ale tyto metody se ukázaly být optimálními ve 2D jak pro nejhorší případ, tak pro případ očekávaný. Literatura uvádí většinou jen metody pro dvojrozměrný případ, protože slučovací fáze je „jednoduchá“ jen v E^2.

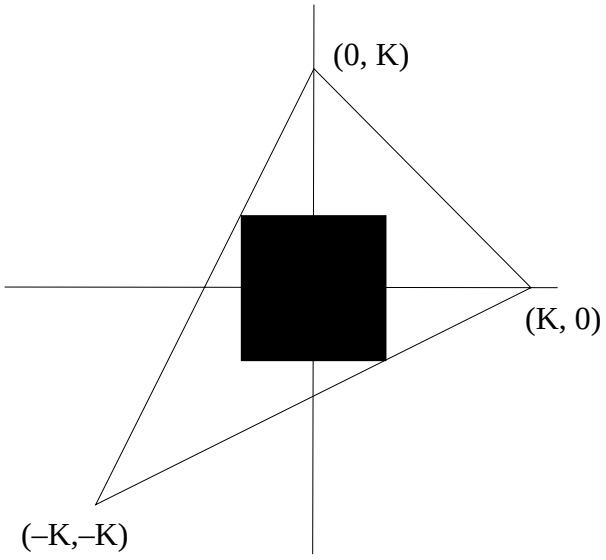
3 Inkrementální Delaunayova triangulace

Ačkoliv algoritmus konstrukce *DT* inkrementálním vkládání má složitost $O(n^2)$ v nejhorším případě a $O(n \cdot log(n))$ očekávanou časovou složitost, je velmi populární díky své jednoduchosti a robustnosti. Algoritmus umožňuje modifikaci pro CDT, ne-Euklidovskou metriku a je velmi podobný jak pro 2D tak i pro 3D.

3.1 Volba počátečního simplexu

Konstrukce *DT(S)* je zahájena vytvořením konvexní obálky *CH*(S) nebo vytvořením dočasného velkého trojúhelníku (resp. tetrahedronu), uvnitř něhož se nacházejí všechny vstupní body množiny S. Druhá varianta se zdá být výhodnější, protože konstrukce konvexní obálky přidává dodatečný čas a komplikuje algoritmus. Nicméně ani použití počátečního simplexu není zcela bez problémů. Tento simplex musí být vyjmut z *DT(S)* po vložení všech bodů z množiny S, což představuje odstranění všech simplexů, jejichž alespoň jeden vrchol je totožný s vrcholy počátečního simplexu. Navíc otázkou zůstává, jak tento simplex zvolit, tj. jak zvolit souřadnice jeho vrcholů.

Podle [Kol00] by vrcholy měly být dostatečně vzdálené od vkládaných bodů, protože jinak dochází k ovlivnění testu prázdné kružnice (koule) opsané, který je při vkládání používán. Na druhou stranu, pokud jsou souřadnice příliš vzdálené, může dojít k problémům s numerickou stabilitou algoritmu. Souřadnice vrcholů jsou proto odvozeny od min-max boxu. Ve dvourozměrném případě jsou souřadnice (K, 0), (0, K) a (-K, -K), pro trojrozměrný případ pak (K, 0, 0), (0, K, 0), (0, 0, K) a (-K, -K, -K). Hodnota K je stanovena jako desetinásobek velikosti min-max boxu (viz obr. 3.1). Předchozí experimenty ukazují, že ani tato hodnota nemusí být dostatečná a občas po odstranění dočasného simplexu je hranice výsledné triangulace nekonvexním útvarem. Ve dvourozměrném případě se lze tomuto vyhnout nahrazením testu kružnice opsané pro krajní trojúhelníky stabilnějším testem geometrické polohy dvou testovaných trojúhelníků. Toto řešení popisuje [Žal01].



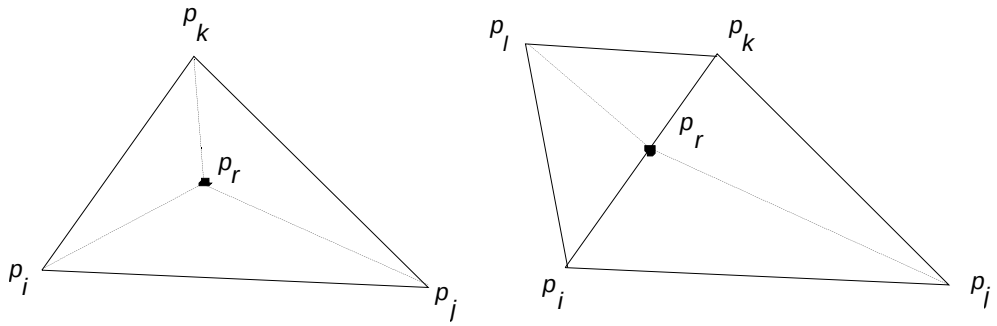
Obr. 3.1: *Volba počátečního simplexu pro dvourozměrný případ. Černě znázorněn min-max box [Kol00].*

3.2 Vložení bodu do triangulace

Pro vložení bodu p_r je nutné nalézt trojúhelník s vrcholy p_i, p_j a p_k (tetrahedron p_i, p_j, p_k a p_l pro trojrozměrnou variantu), se kterým vkládaný bod inciduje. Existuje několik vzájemných poloh „vkládaný bod – simplex“ (pokud je vkládaný bod totožný s vrcholem simplexu, je zanedbán). Pro dvourozměrnou variantu algoritmu se tedy jedná o možnost, že bod p_r leží uvnitř trojúhelníku a možnost, že bod leží na hraně. Ve 3D kromě již zmiňovaných kombinací navíc přibývá možnost, že bod leží ve stěně.

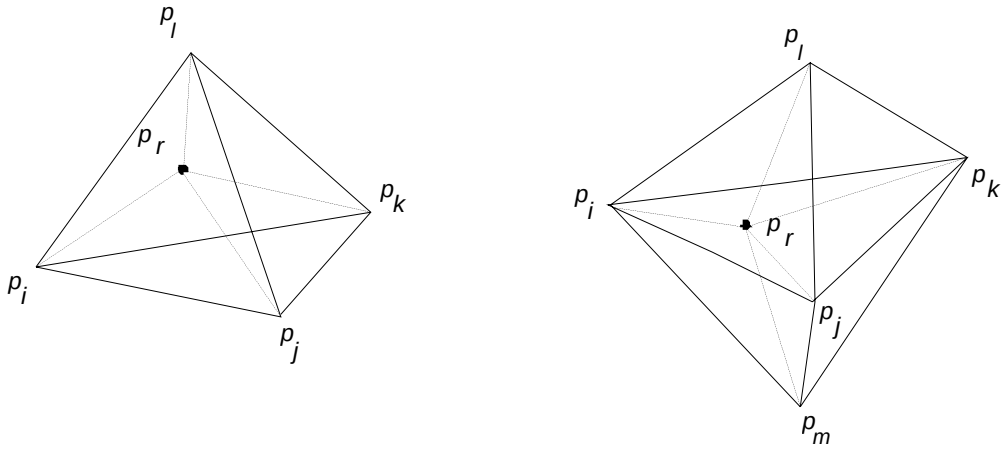
Nejjednodušším případem je varianta: bod se nachází uvnitř simplexu. V tomto případě ze všech vrcholů simplexu jsou zkonstruovány hrany s vkládaným bodem a původní simplex se tak rozpadne na tři nové trojúhelníky (viz obr. 3.2a), nebo ve trojrozměrném případě na čtyři tetrahedrony (viz obr. 3.3a).

Složitější situace nastává pokud bod leží ve dvourozměrném případě na hraně trojúhelníku a v trojrozměrném ve stěně tetrahedronu. Kromě rozdělení simplexu p_i, p_j a p_k (resp. p_i, p_j, p_k a p_l) musí dojít též k rozdělení sousedního simplexu, který s tímto simplexem sdílí hranu (resp. stěnu), na které se vkládaný bod p_r nachází. Výsledkem jsou čtyři trojúhelníky (viz obr. 3.2b), nebo ve 3D šest tetrahedronů (viz obr 3.3b).



a) Bod leží uvnitř trojúhelníku b) Bod leží na hraně

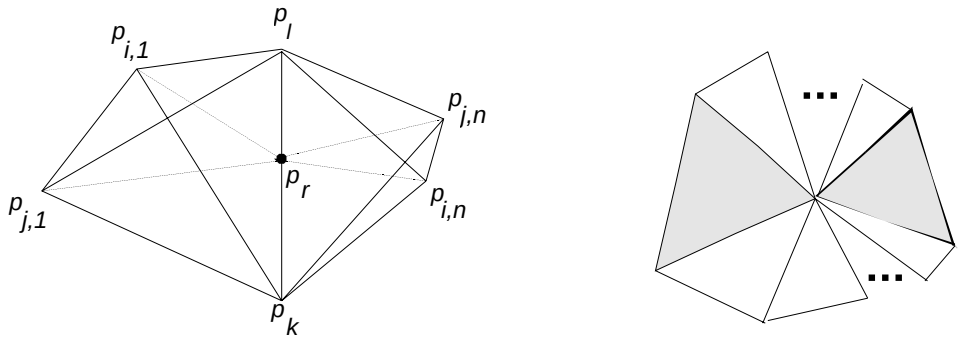
Obr. 3.2: *Vložení bodu do DT ve 2D.*



a) Bod leží uvnitř tetrahedronu b) Bod leží na stěně

Obr 3.3: *Vložení bodu do DT ve 3D.*

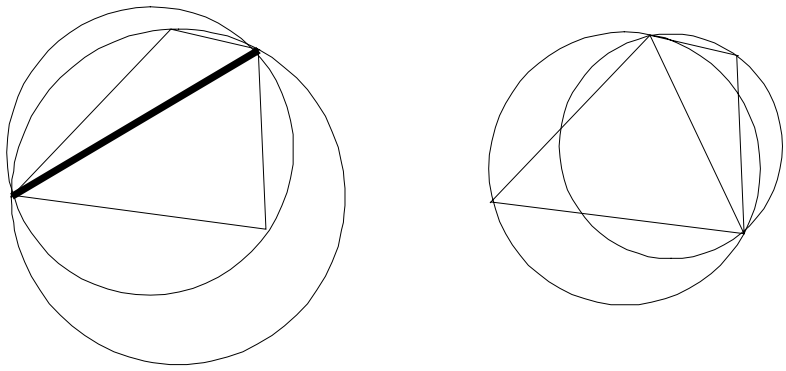
Nejsložitějším případem je varianta, kdy bod ve 3D verzi leží na hraně. V tomto případě totiž musí být rozděleny tetrahedrony, které obsahují tuto hranu. Při nejhorším je nutné rozdělit všechny tetrahedrony v *DT*. Každý tetrahedron se rozdělí na dva nové (viz obr. 3.4). Důsledkem je, že zatímco pro dvourozměrnou triangulaci lze stanovit velikost vzniklé trojúhelníkové sítě na základě znalosti počtu vstupních bodů a počtu bodů *CH(S)*, není možné tentýž výpočet uskutečnit v 3D verzi.



Obr. 3.4: *Vložení bodu na hranu ve 3D. Pro přehlednost jsou zobrazeny pouze dva tetrahedrony incidující s hranou. Vpravo pak průmět do roviny xy.*

3.3 Legalizace

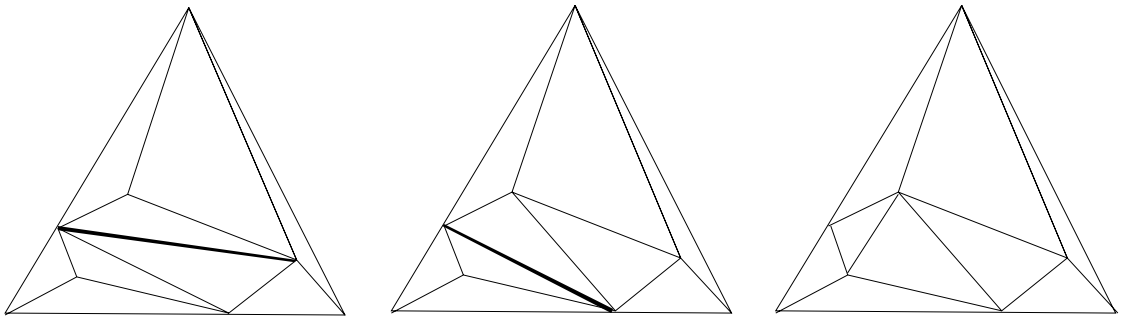
Vložením nového bodu do Delaunayovy triangulace však není zaručeno, že nově vzniklá triangulace bude opět Delaunayovská. Všechny vnější hrany (resp. stěny ve 3D) nově vzniklých simplexů musí být proto otestovány, jestli neporušují kritérium prázdné kružnice (koule) opsané, tj. zda jsou legální. Testuje se, jestli vrchol sousedního simplexu neleží uvnitř kružnice (koule) opsané každému novému simplexu. Je-li test úspěšný, testování tímto směrem nepokračuje. Pokud test není úspěšný, triangulace musí být opravena. Oprava (legalizace) se provádí aplikací lokálních transformací.



Obr. 3.5: *Test prázdné kružnice opsané. Původní triangulace (vlevo) nesplňující test je převedena prohozením hrany na Delaunayovskou [Kol00].*

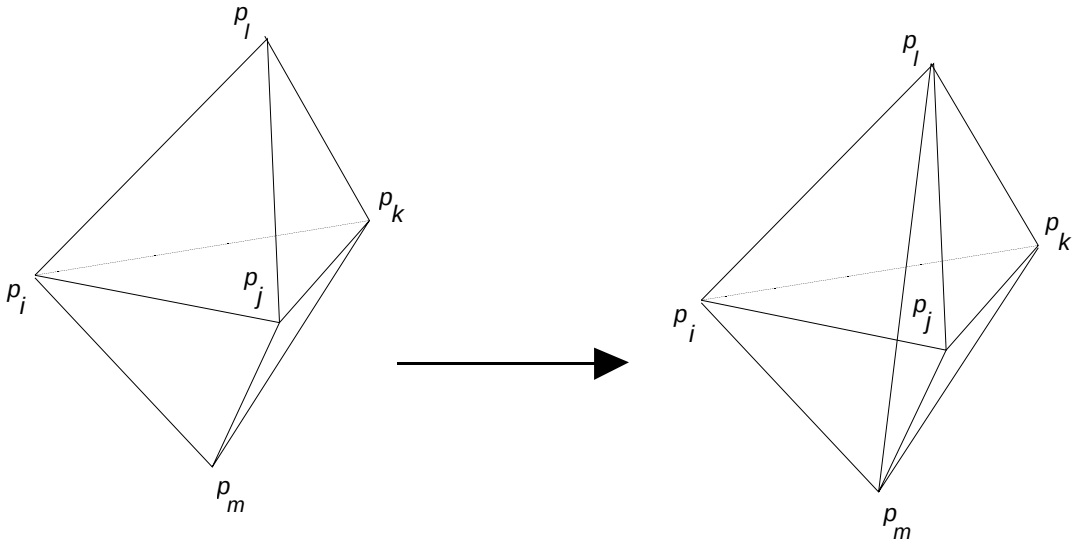
Pro dvourozměrnou verzi je legalizace jednoduchá. Hrana je jednoduše prohozena (viz obr 3.5). Je zřejmé, že prohozením hrany se v síti objeví nové dva trojúhelníky a ověřování platnosti hran musí pokračovat (opět se musí ověřit všechny vnější hrany). Vložením bodu do

triangulace může tedy dojít ke změně celé triangulace. Na obrázku 3.6 je ukázán případ, kdy po prohození hrany vede k nutnosti prohodit další hranu, tj. k propagaci prohazování.



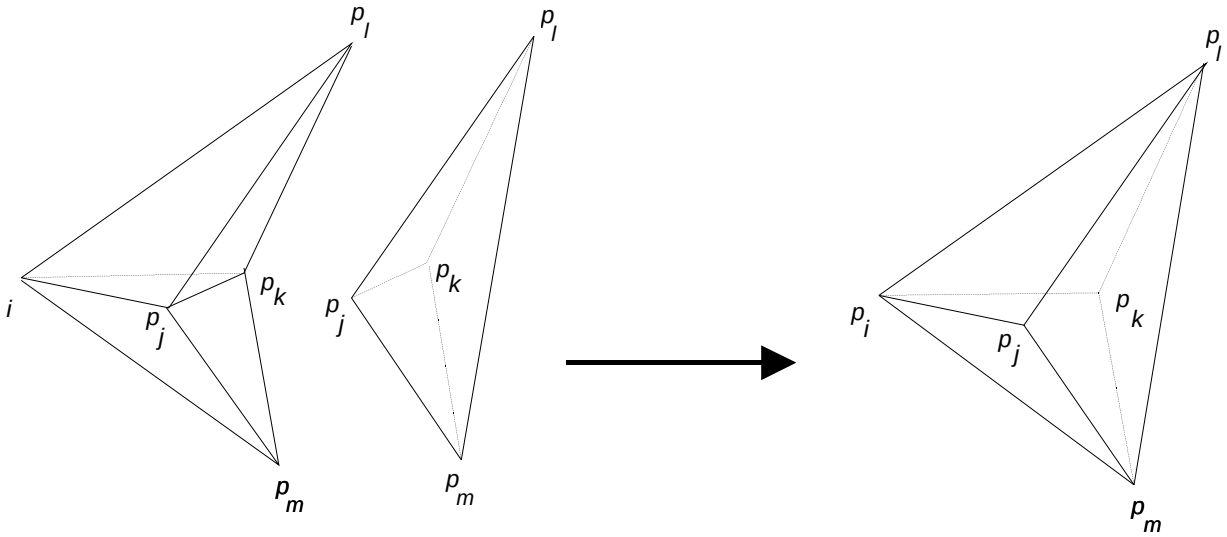
Obr. 3.6: *Propagace prohazování hran ve 2D. Čárkovaně znázorněno rozdělení, které změny vyvolalo, tučně pak nelegální hrana, která musí být prohozena.*

Oprava triangulace ve 3D není již tak jednoduchá a spočívá v aplikaci lokální transformace [Joe91] na soubor dvou až čtyř tetrahedronů. Mějme dvojici tetrahedronů s vrcholy p_i, p_j, p_k, p_l a p_m se společnou nelegální stěnou $p_i p_j p_k$. Pokud spojnice $p_l p_m$ prochází touto stěnou a současně neprotíná některou z jejích hran, potom tyto dva tetrahedrony jsou nahrazeny trojicí čtyřstěnů $p_i p_j p_l p_m$, $p_i p_k p_l p_m$ a $p_j p_k p_l p_m$ (viz obr 3.7).



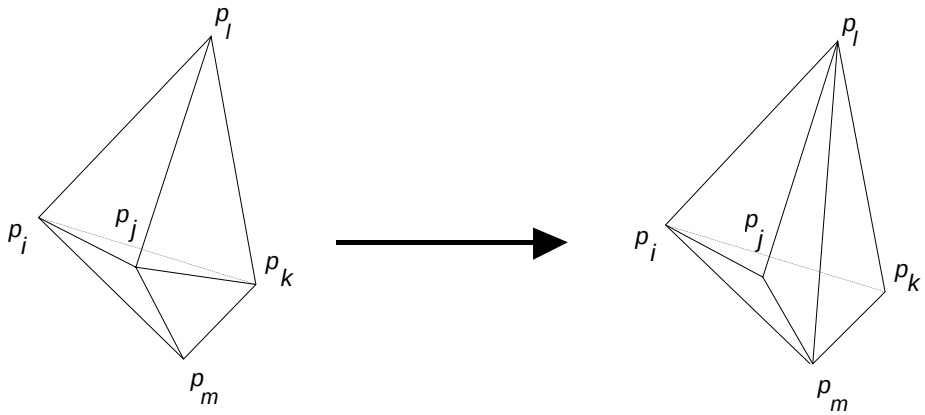
Obr. 3.7: *Transformace dvou sousedních tetrahedronů $p_i p_j p_k p_l$ a $p_i p_j p_k p_m$ s nelegální stěnou $p_i p_j p_k$ na tři tetrahedrony $p_i p_j p_l p_m$, $p_i p_k p_l p_m$ a $p_j p_k p_l p_m$.*

Pokud spojnice $p_l p_m$ hranou neprochází a současně protíná pouze jeden další tetrahedron a navíc tento tetrahedron doplňuje prostor tak, jak je naznačeno na obrázku 3.8, jsou tyto tři tetrahedrony nahrazeny dvojicí tetrahedronů (viz obr 3.8).



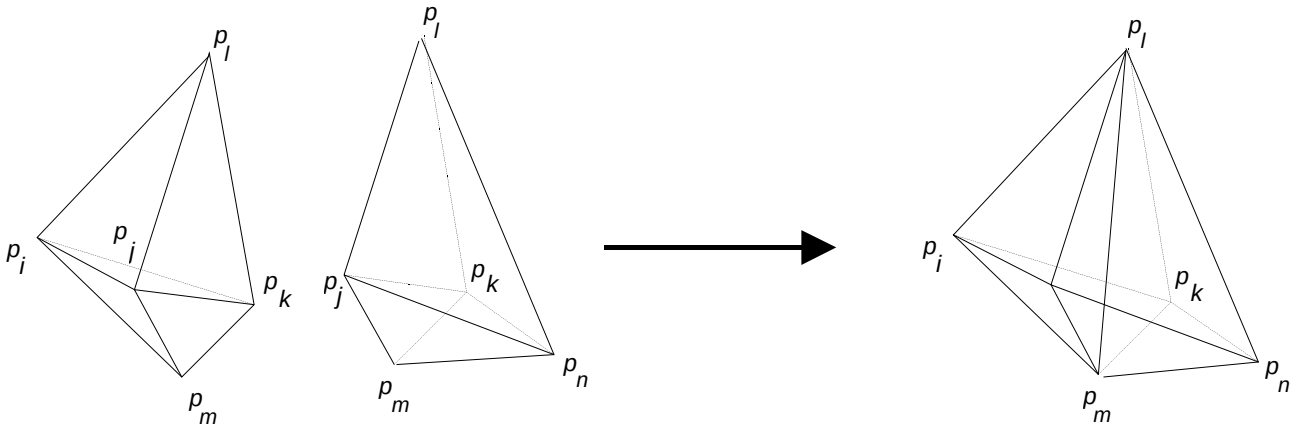
Obr. 3.8: *Transformace tří sousedních (znázorněn tetrahedron a $p_j\,p_k\,p_l\,p_m$ odděleně pro větší přehlednost) tetrahedronů $p_i\,p_j\,p_k\,p_l$, $p_i\,p_j\,p_k\,p_m$ a $p_j\,p_k\,p_l\,p_m$ na dvojici tetrahedronů $p_i\,p_j\,p_l\,p_m$ a $p_i\,p_k\,p_l\,p_m$.*

Poněkud komplikovanější je situace, kdy spojnice protíná např. hranu $p_j\,p_k$ nelegální stěny $p_i\,p_j\,p_k$, tzn. stěna $p_j\,p_k\,p_l$ leží ve stejné rovině jako stěna $p_j\,p_k\,p_m$. Pokud bychom dvojici tetrahedronů transformovali podle předchozího postupu na trojici čtyřstěnů, byl by objem jednoho z nich nulový, což je nepřípustné. Proto tato dvojice bude transformována jednoduchou záměnou stěny $p_i\,p_j\,p_k$ za $p_i\,p_l\,p_m$ (viz obr 3.9).



Obr. 3.9: *Transformace dvou sousedních tetrahedronů se stěnami $p_j\,p_k\,p_l$ a $p_j\,p_k\,p_m$ v jedné rovině na dvojici tetrahedronů $p_i\,p_j\,p_l\,p_m$ a $p_i\,p_k\,p_l\,p_m$.*

Pokud však koplanární (v jedné rovině) stěny nejsou vnějšími stěnami triangulace, potom existuje sousední dvojice tetrahedronů. Uvedenou záměnou stěny vznikne kombinace porušující definici Delaunayovy triangulace, protože dva sousední simplexy nyní sdílejí pouze část stěny. Proto je nutné provést záměnu stěny i v sousední dvojici tetrahedronů. Tato operace tedy nahrazuje čtveřici tetrahedronů jinou čtveřicí (viz obr 3.10).



Obr. 3.10: *Transformace čtyř sousedních (znázorněno odděleně pro větší přehlednost) na čtveřici tetrahedronů $p_i p_j p_l p_m$, $p_i p_k p_l p_m$, $p_j p_l p_m p_n$ a $p_k p_l p_m p_n$.*

3.4 Vyhledání simplexu

Klíčovým problémem je rychlé nalezení simplexu, který inciduje s vkládaným bodem. Jednou z možností, jak toho docílit, je použití orientovaného acyklického grafu (*DAG*) [Ber97]. Díky podobnosti se stromem budeme používat stromovou terminologii (včetně pojmů kořen, vnitřní uzel a list). Každý uzel reprezentuje simplex. Rozdělením simplexu nebo aplikací lokální transformace získá uzel tolik synů, kolik nových simplexů bylo vytvořeno. Platná *DT* je tedy uložena v listech struktury, zatímco počáteční simplex je reprezentován kořenovým uzlem. Vyhledávání simplexu v DAG může být provedeno v očekávaném případě se složitostí *O(log n)* a *O(n)* v případě nejhorším. Za předpokladu, že pořadí vkládání vstupních bodů do triangulace je randomizováno, pravděpodobnost výskytu nepříznivého případu je velmi nízká, protože „strom“ je s vysokou pravděpodobností vyvážený (randomizace zde nahrazuje vyvažování stromu). Příklad, jak změna struktury probíhá, znázorňuje obr. A.1. Vlastní algoritmus je k nalezení v A.2 a A.3.

Dalším oblíbeným způsobem (zejména v E^2) jsou nejrůznější modifikace strategie procházek, která spočívá v tom, že simplex, který obsahuje vkládaný bod je nalezen navštívením simplexů mezi vkládaným bodem a nějakým počátečním bodem [Gui85]. Vyhledání simplexu bývá provedeno se složitostí $O(n^{1/3})$ v 2D a $O(n^{1/4})$ v 3D.

Neposlední možností je využití quadtree a „bucketing techniky“ [Žal01].

4 Paralelní algoritmy pro konstrukci Delaunaovy triangulace

Flynn klasifikoval výpočetní systémy do tří skupin. SISD – Single Instruction Single Data představuje systémy s klasickou von Neumannovou architekturou, tj. jeden proud instrukcí (program) zpracovává jeden proud dat.

SIMD – Single Instruction Multiple Data zahrnuje vektorové a maticové procesory, ve kterých se provádí tatáž instrukce současně na každém procesoru, ale pro různá data. Tyto systémy jsou vhodné pro náročné numerické výpočty, ale z hlediska programátora se díky transparentnosti paralelismu nikterak neliší od systémů předchozí třídy.

Poslední třída MIMD – Multiple Instruction Multiple Data obsahuje všechny výpočetní systémy s více procesory (tj. multiprocesory), každý procesor zpracovává svým nezávislým proudem instrukcí svá data. Multiprocesory [Jež99] kategorizuje na multiprocesory se sdílenou pamětí a s distribuovanou pamětí.

Multiprocesory se sdílenou pamětí jsou většinou tvořeny rovnocennými (stejná možnost práce, tj. např. stejný přístup ke sdíleným periferním prostředkům) homogenními procesory (stejný typ, stejná množina instrukcí). Programy psané pro tuto paralelní architekturu jsou relativně jednoduché, většinou postačuje jediný kód pro všechny procesory, který může běžet jak na *k* procesorech, tak i na *k*+1 procesorech bez nutnosti změny zdrojového textu a opětovného sestavení programu. Procesy mezi sebou komunikují prostřednictvím sdílené paměti, operační systém poskytuje prostředky pro jejich synchronizaci. S rostoucím počtem procesorů v systému narůstá zatížení sběrnice a tudíž klesá i efektivita paralelního výpočtu. Proto multiprocesory se sdílenou pamětí obsahují nejvýše několik desítek výpočetních jednotek.

Multiprocesory s distribuovanou pamětí jsou tvořeny procesory s vlastní lokální pamětí, propojené mezi sebou sériovými linkami. Nejpoužívanější topologií je propojení do cyklicky uzavřené dvourozměrné mřížky (každý procesor je propojen se čtyřmi sousedními) nebo do obecnější více rozměrné mřížky označované n-CUBE (každý procesor propojen s 2^n procesory). Od počítačových sítí se odlišují zejména geometrií (vzdálenost procesorů řádu cm nebo desítky cm), jednoduchým komunikačním protokolem a vysokým stupněm integrace řízení (jeden OS, sdílené periférie). Procesy mezi sebou komunikují zasíláním zpráv po sériových linkách. Tato paralelní architektura umožňuje vysoký stupeň paralelismu, stovky či tisíce procesorů, za což zaplatí programátor tím, že jeho program není univerzální pro libovolnou topologii a ani pro různý počet zapojených procesorů. Protože multiprocesor se sdílenou pamětí nepatří mezi hardware, který si může téměř kdokoliv „dovolit“, budeme v textu této práce nazývat tuto architekturu speciální.

V následujících podkapitolách uvedeme pro jednotlivé přímé metody konstrukce *DT*, které byly popsány v druhé kapitole, nejznámější paralelní algoritmy. Je nutné podotknout, že některé metody jsou svou podstatou natolik sekvenční, že autorovi není znám žádný jejich paralelní zástupce.

4.1 Inkrementální vkládání

Paralelizace konstrukce *DT* inkrementálním vkládáním není jednoduchá, protože vložení bodu může modifikovat celou triangulaci. Není proto divu, že tato třída není příliš zastoupena.

4.1.1 Paralelní Bowyer-Watsonův algoritmus

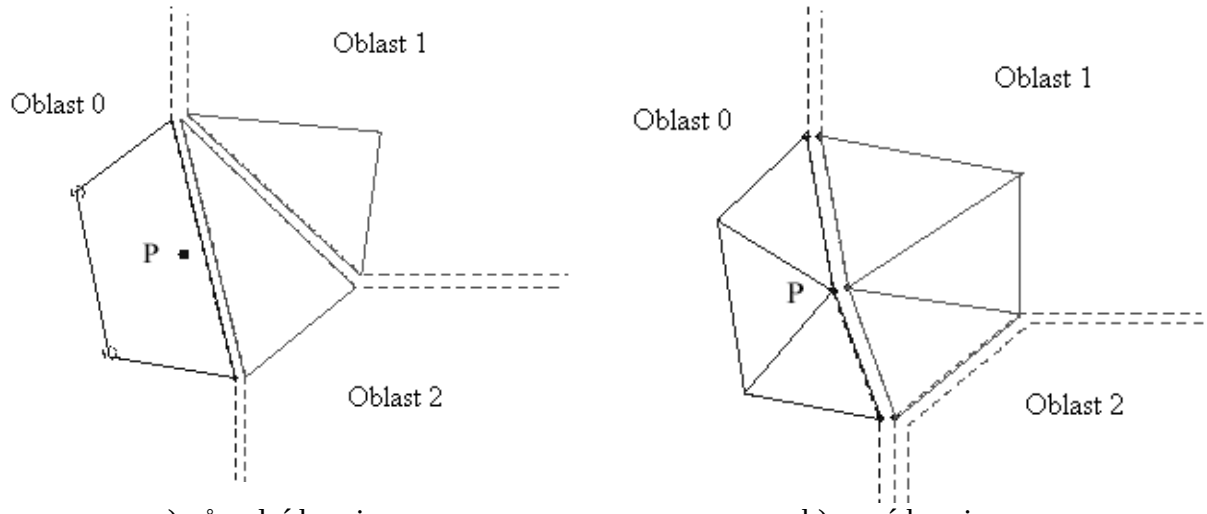
Watsonův algoritmus se poněkud liší od schématu popsaného v kapitole pojednávající o inkrementálním vkládání. Právě tato odlišnost umožňuje paralelizaci algoritmu [Chr96]. Jedná se o to, že při legalizaci se odstraní všechny nelegální hrany z triangulace a vzniklá po-

lygonální oblast (díra), uvnitř které leží posledně vložený bod p_r, se opětovně triangularizuje tak, že vrcholy oblasti jsou spojeny s bodem p_r. Diagonály jsou pak přeházeny podle principu lokálního vylepšování tak, aby výsledná triangulace oblasti byla Delaunayova.

Paralelní verze algoritmu prezentovaná v [Chr96] zahajuje sekvenční konstrukcí dílčí Delaunayovy triangulace. Simplexy výsledné triangulace jsou rozděleny na *k* souvislých částí, kde *k* je počet procesů. Každému procesu je přidělena jedna oblast. Do té postupně přidává body. Po vložení bodu je stanovena díra, která se má retriangularizovat. Pokud se nachází na území přiděleném pouze jednomu procesu, může proces operaci dokončit nezávisle na činnosti ostatních procesů. V opačném případě se musí synchronizovat všechny procesy, do jejichž území díra zasahuje. Vlastní triangulaci pak provede proces, který bod vkládá.

Retriangulací díry na hranici se změní hranici mezi oblastmi procesů (viz obr. 4.1). Nejjednodušším způsobem by bylo nově vzniklé simplexy přidělit jednomu procesu, ale tento způsob by vedl k nevyváženosti zatížení. [Chr96] proto používá heuristiku, která nejen, že dbá na vyváženost počtu simplexů v každé oblasti, ale také minimalizuje délku hranice. Druhá vlastnost je velmi významná, protože čím kratší hranice mezi oblastmi je, tím méně často bude díra zasahovat do více oblastí, a proto počet synchronizací bude minimalizován.

[Chr99] uvádí ničím nepodložené tvrzení, že urychlení paralelní verze je téměř lineární. Namísto srovnání sériové a paralelní verze však srovnává čtyři různé heuristiky.



a) původní hranice b) nová hranice

Obr. 4.1: *Díra vzniklá díky vložení bodu P do triangulace zasahuje do tří oblastí, hranice mezi nimi se po zasynchronizování a retriangulaci změní. [Chr99]*

4.2 Inkrementální konstrukce

Díky podstatě inkrementální konstrukce paralelizace těchto algoritmů je relativně snadná. Proto také je tato třída hojně zastoupená.

Často se využívá vlastností speciálních paralelních architektur. [Mag98] zmiňuje algoritmus pro konstrukci *2D DT*, který používá „Orthogonal Tree Network“. Tato architektura sestává z pole $\sqrt{N} x \sqrt{N}$ procesorů. Každá buňka pole je listem binárního stromu, v jehož vnitřních uzlech i v kořeni se nacházejí další procesory. Ty se starají o komunikaci mezi listy, zatímco procesory v listech provádějí vlastní výpočet. Paralelní časová složitost zmiňovaného algoritmu je $O(\log^2 N)$, kde N je počet vstupních bodů.

4.2.1 Leeův algoritmus

Zajímavý algoritmus, tentokrát pro konstrukci *CDT* ve dvourozměrném případě, uvádí [Lee97]. Opět je využita speciální paralelní architektura. Jedná se o superpočítač Intel Paragon, který má až dva tisíce stejných samostatných procesorů. Každý procesor má k dispozici minimálně 16MB lokální paměti a může na něm běžet mnoho procesů, přičemž každý proces může mít mnoho vláken. Plánovací mechanismem je standardní OSF/1.

Jednotlivé procesory jsou propojeny rychlou sítí přes *message routing chip* (MRC). Tyto jednotky jsou užívány pro odesílání a příjem zpráv připojeného procesoru a umožňují komunikaci každý s každým.

Tento algoritmus je třífázový. Nejprve je sestrojena paralelně obyčejná Delaunayova triangulace, poté jsou odstraněny všechny hrany, které protínají povinné hrany, a nakonec jsou přidány povinné hrany a případné existující díry retriangularizovány. Tento postup sice umožňuje začlenění povinných hran, ale neumožňuje nekonvexní tvary *DT*. Protože tato práce není věnována problematice „*Constrained DT*“, uveďme jen algoritmus pro konstrukci *DT* (tj. první fázi Leeova algoritmu):

1. Každému procesoru je zaslána množina vstupních bodů.
2. Každý procesor pro všechny body v_i jemu příslušné nalezne k bodu v_i nejbližší bod v_j a sestrojí hranu e_{ij}.
3. Každý procesor ke každé hraně, kterou sestrojil v předchozím kroku nalezne nejbližší bod v levé polorovině a nejbližší v pravé polorovině a sestrojí dva trojúhelníky.
4. Procesor 1 sesbírá všechny nově vytvořené objekty, odstraní redundantní hrany (tj. hrany, které byly sestrojeny vícekrát) a rozešle zbylé hrany mezi všechny procesory.
5. Každý procesor ke každé obdržené hraně sestrojí dva trojúhelníky podle bodu 3.
6. Kroky 4 a 5 se opakují, dokud jsou v bodě 4 generovány nové objekty.

Algoritmus předpokládá stejný počet procesorů jako je počet vstupních bodů. Pak platí, že v bodě 2 je sestrojena jediná hrana, a navíc lze ukázat, že kroky 4 a 5 se opakují nejvíce třikrát. Složitost uvedeného algoritmu je tedy *O(N)*. Poznamenejme, že v případě nižšího počtu procesorů složitost stoupá.

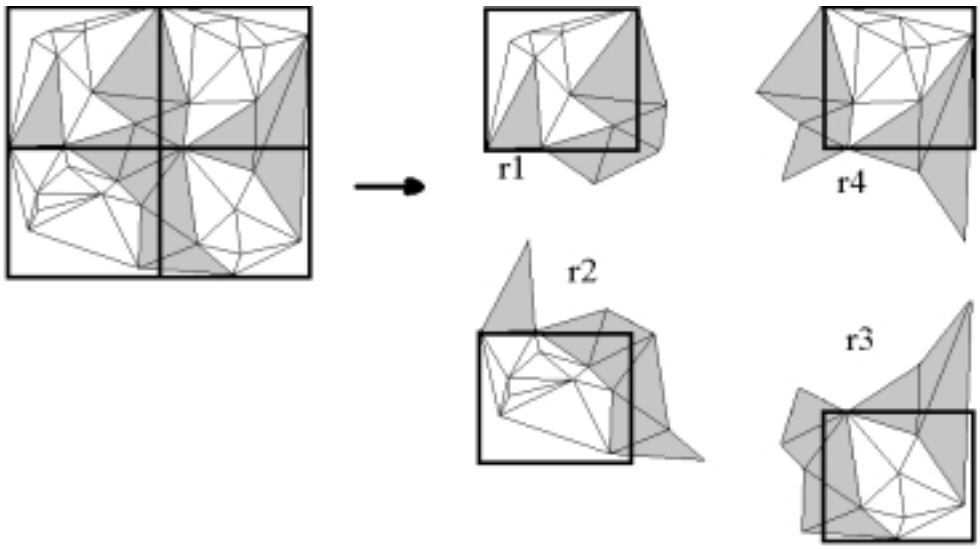
[Lee97] bohužel neuvádí experimentální výsledky svého algoritmu.

4.2.2 Algoritmus InCoDe

Algoritmus prezentovaný [Cig93] nazvaný Incremental Construction Delaunay je relativně jednoduchý. Kvádr (resp. obdélník ve 2D) ohraničující vstupní body množiny *S* je rozdělen na podoblasti. Každá z těchto podmnožin je přidělena jednomu procesoru, a ten provede triangulaci metodou inkrementální konstrukce.

Algoritmus využívá seznam aktivních stěn *AFL* (datová struktura bývá reprezentována jako hashovací tabulka), který obsahuje ty stěny, pro něž ještě nebyl sestrojen sousední simplex. Na začátku je tento seznam prázdný, po konstrukci prvního simplexu jsou do něj přidány všechny stěny tohoto simplexu. Algoritmus vybírá stěny postupně ze seznamu a nalezne k nim bod, který zkonstruuje další Delaunayovský simplex. Stěny simplexu, které se v *AFL* dosud nenacházejí, jsou do něj vloženy. Stěny simplexu, které se naopak v *AFL* již nacházejí, jsou z něj odstraněny. Činnost končí v okamžiku, kdy *AFL* je prázdný.

Trik algoritmu spočívá v tom, že do seznamu se vkládají pouze stěny (resp. hrany) ležící zcela v přidělené oblasti. Každý procesor sestrojí proto triangulaci pouze své oblasti s tím, že hraniční simplexy jsou zkonstruovány více procesory současně (viz obr. 4.2). Duplicitní simplexy jsou odstraněny v sekvenční závěrečné fázi.



Obr. 4.2: *Příklad triangulace ve 2D, hraniční trojúhelníky (šedivé) jsou zkonstruovány duplicitně více procesy.*

Urychlení algoritmu pro konstrukci *3D DT* uváděné v [Cig93] je pro uniformní data o 20000 bodech 1.79 – 19.01 pro 2 – 64 PEs na počítači nCUBE 2 system model 6410. Zatímco hodnota 1.79 pro dva procesory je přijatelná, hodnota 19.01 pro 64 procesorů je velmi nízká. Je to způsobené jednak narůstajícím počtem duplicitních simplexů a jednak nerovnoměrnou zátěží procesorů, každá oblast obsahuje různý počet vstupních bodů. [Har97] podotýká, že v případě neuniformních dat je výsledný čas algoritmu až 10krát vyšší. Nespornou výhodou algoritmu však je jeho jednoduchost a použitelnost jak pro 2D tak i 3D.

4.2.3 Tengův algoritmus

Algoritmus popisovaný v [Ten93] se příliš neliší od předchozího algoritmu. Zásadní rozdíl je v tom, že seznam stěn je společný pro všechny procesy, což sice vede nutně k synchronizaci procesů (částečně je eliminována použitím přídavné struktury), ale odpadá sekvenční fáze potřebná v InCoDe pro odstranění duplicit. Publikované urychlení 3D algoritmu na počítači CM-5 pro uniformní data o 16000 bodech je 3.43 pro 128 PEs a 6.08 pro 256 PEs, přičemž urychlení je počítané vůči době výpočtu pro 32 PEs.

4.3 Rozděl a panuj (D&C)

Strategie rozděl a panuj je svou podstatou vhodná k paralelizaci, a proto také tato třída algoritmů je nejobsáhlejší. Do této třídy patří všechny algoritmy, které používají princip Rozděl a panuj, bez ohledu na způsob, kterým je provedena lokální triangulace.

Klasická D&C obsahuje dvě fáze, a to rozdělení vstupní množiny a sloučení výsledných triangulací, při kterých je vyžadována synchronizace procesorů. Kritickou fází, která silně ovlivňuje efektivitu těchto algoritmů, je zejména slučování vytvořených triangulací. Rovněž implementace této fáze není příliš jednoduchá. Kromě problémů se synchronizací, je častým problémem D&C algoritmů nevyváženost zatížení jednotlivých procesorů.

4.3.1 Hardwick – Blellochův Algoritmus

[Har97] popisuje poněkud složitý algoritmus pro konstrukci *2D DT* založený na [Ble96], který nevyžaduje speciální paralelní architekturu. Algoritmus využívá ke své činnosti Machiavelli Toolkit. Jedná se o programové vybavení implementované v C, jež používá MPI komunikační protokol. Proto je navržený algoritmus použitelný jak na paralelních architekturách se sdílenou pamětí, tak i na architekturách s distribuovanou pamětí.

Tento algoritmus může být mylně řazen do třídy algoritmů využívajících vyšší dimenzi. Algoritmus sice vyšší dimenzi používá, ale ne pro vlastní konstrukci Delaunayovy triangulace, nýbrž pro optimální dělbu množiny vstupních bodů. Navíc se jedná o rekurzivní algoritmus, který vychází ze strategie D&C.

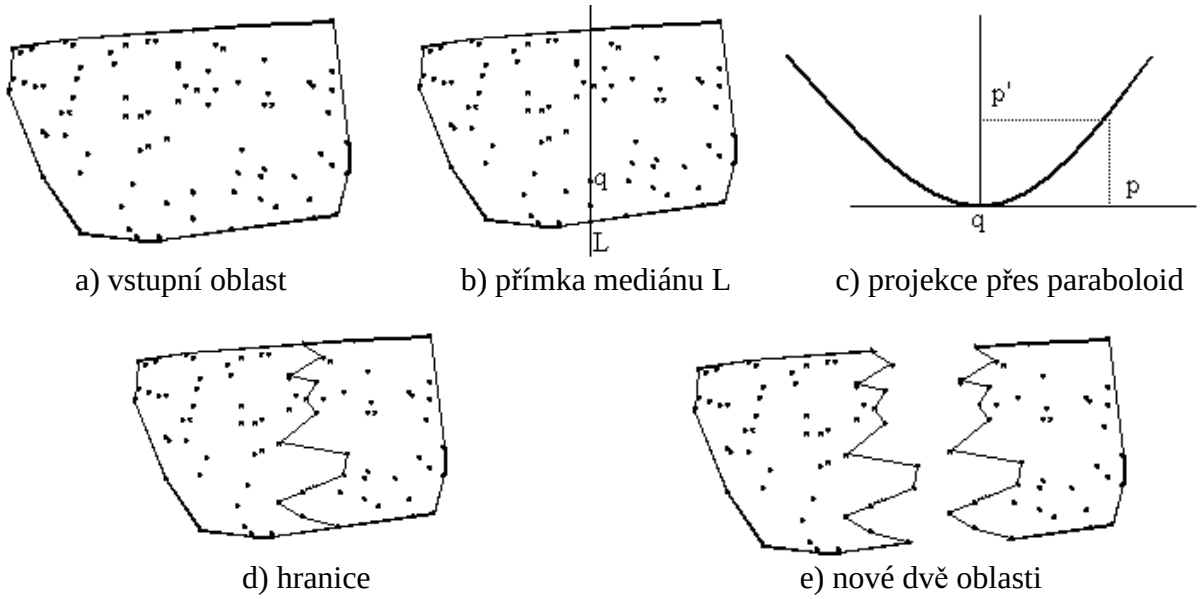
Popišme si nyní jeho jednotlivé fáze (viz algoritmus 4.1 – funkce PARDEL). Fáze rozdělení je nejsložitější částí celé konstrukce *DT*. Vstupní množina bodů v rovině *xy*, které se nachází buď uvnitř anebo na hranici (obecně nekonvexní) zpracovávané oblasti, je rozdělena v této fázi na dvě přibližně stejné části způsobem popsaným v následujícím odstavci.

```
PARDEL(S,B,T)
In: množina vstupních bodů S, množina hran B, které tvoří hranici regionu
  obsahujícího všechny body z S, T počet volných procesorů.
Out: Delaunayova triangulace regionu z B.

  1. if T == 1 return SERIAL(S,B)
  2. S' = množina transformovaných bodů (promítnutých přes paraboloid do
     příslušné roviny) z množiny S
  3. H' = LOWER_CONVEX_HULL(S')
  4. H = hranice po promítnutí H' zpět do roviny xy
  5. S^L = množina bodů ležící vlevo od H
  6. S^R = množina bodů ležící vpravo od H
  7. B^L = hranice oblasti vlevo od H
  8. B^R = hranice oblasti vpravo od H
  9. Return PARDEL(S^L, B^L, T/2) ∪ PARDEL(S^R, B^R, T/2)
```

Algoritmus 4.1: *Rekurzivní funkce ve smyslu D&C pro výpočet 2D DT využívající konstrukce konvexní obálky ve vyšší dimenzi.*

Nejprve se vybere bod, jehož hodnota souřadnice *x* je mediánem všech souřadnic *x* vnitřních bodů (body na hranici se tohoto procesu neúčastní), poté se tímto bodem proloží rovina *xz* a současně se zkonstruuje paraboloid s vrcholem ležícím na průsečíku rovin *xy* a *xz* (průsečíkem je přímka *L*) a jehož osa je kolmá na rovinu *xy*. Všechny vstupní body jsou promítnuty na povrch paraboloidu a odtud na sestrojenou rovinu *xz*. Pro promítnuté body je sestrojena dolní polovina konvexní obálky některým efektivním algoritmem (rutina LOWER_CONVEX_HULL); [Ble96] používá paralelní verzi Overmars a Van Leeuwenova algoritmu. Hrany konvexní obálky jsou promítnuty přes paraboloid zpět do roviny xy. Vznikne tak „klikatá“ hranice podél přímky *L*, která rozděluje vstupní množinu bodů na dvě přibližně stejné části (co se týče počtu bodů) a navíc se dá prokázat, že elementy této hranice jsou Delaunayovy hrany budoucí triangulace. Názorně je tato fáze zobrazena na obr. 4.3.



a) vstupní oblast b) přímka mediánu L c) projekce přes paraboloid

d) hranice e) nové dvě oblasti

Obr. 4.3: *Činnost algoritmu 4.1*

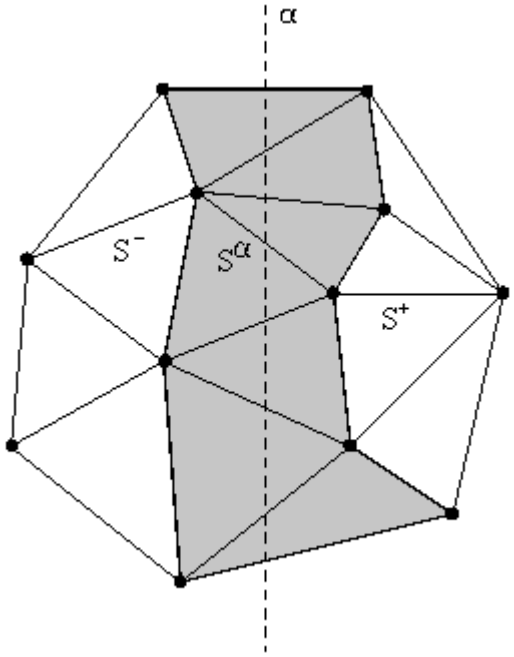
Rekurzivní dělení končí v okamžiku, kdy pro zpracovávanou oblast je k dispozici již jen jediný procesor a tato obecně nekonvexní oblast je triangularizována libovolným sériovým algoritmem. [Har97] jako lokální algoritmus pro *DT* doporučuje Dwyerův [Dwy86] sériový algoritmus, který patří mezi nejrychlejší.

Uvedený algoritmus je implementačně poněkud složitý a navíc nerozšiřitelný do vyšší dimenze, ale má dvě přednosti. Medián zajišťuje přibližně stejně veliké oblasti a tudíž i téměř rovnoměrné zatížení procesorů. Konstrukce konvexní obálky eliminuje obtížnou slučovací fázi standardních D&C algoritmů, protože spoj je sestaven přednostně. Díky těmto přednostem je urychlení Hardwickovy implementace vůči sériové verzi pro uniformní data o 128K bodů na počítači SGI Power Challenge se sdílenou pamětí asi 1.8 – 5.8 pro 2 – 8 PEs a na počítači IBM SP2 s distribuovanou pamětí asi 1.5 – 4.0 pro 2 – 8 PEs.

4.3.2 Algoritmus DeWall

Delaunay Wall (DeWall) algoritmus [Cig93] je založen na principu existence nadroviny α, která rozděluje prostor na tři množiny, a to na oblast S^-, ve které jsou všechny Delaunayovy simplexy vlevo od nadroviny, na oblast S^+, ve které jsou všechny Delaunayovy simplexy vpravo od nadroviny a na oblast obsahující simplexy, kterými nadrovina prochází, označovanou S^α.

Volbou nadroviny α se rozdělí množina vstupních bodů P na dvě podmnožiny označované P^- a P^+. Algoritmus nejprve sestrojí inkrementální konstrukcí oblast S^α, poté je aplikován rekurzivně na množinu bodů P^- a sestrojí (opět inkrementální konstrukcí) oblast S^-. Rekurzivní aplikace je provedena rovněž pro množinu bodů P^+ a po ní se sestrojí oblast S^+. Výsledná triangulace je dána pouhým sdružením těchto tří oblastí, tzn., že tento algoritmus opět eliminuje náročnou slučovací fázi (viz obr. 4.4).



Obr. 4.4: *Triangulace ve 2D [Mag98]*.

Ke své činnosti používá DeWall tři disjunktní seznamy stěn (resp. hran ve 2D verzi) sestrojených čtyřstěnů (resp. trojúhelníků). Pro efektivní práci s položkami těchto seznamů jsou tyto seznamy implementovány jako hashovací tabulky. Tabulka označená AFL^+ obsahuje všechny stěny (resp. hrany), které jsou celé vpravo od dělicí nadroviny α, tabulka označená AFL^- obsahuje všechny stěny (resp. hrany), které jsou celé vlevo od dělicí nadroviny α, a je zřejmé, že poslední tabulka AFL^α tedy zahrnuje hrany, které protínají dělicí nadrovinu. Činnost sériové verze DeWall popisuje algoritmus A.4.

Paralelizace uvedeného algoritmu i přes jeho podstatu není triviální. Přirozeným řešením by bylo v bodě rekurzivní aplikace vytvořit nové dva procesy, každý pro jednu vytvořenou podmnožinu (resp. jeden proces, který obdrží P^- zatímco aktivní proces bude počítat *DT* nad P^+). Oba procesy paralelně spočtou DeWall. Každý tento proces vytvoří opět další procesy, tj. výpočtu v další úrovni se zúčastní již čtyři procesy. [Cig93] nedoporučuje tento způsob kvůli dodatečné režii vytvoření nového procesu a navrhuje metodu, která předpokládá jednorázové založení *k* procesů v samém počátku algoritmu.

Každý proces, který běží podle algoritmu 4.3, má přiřazen identifikátor „řídící“ jeho činnost. Načtenou množinu vstupních bodů rozdělí na dvě skupiny, vybere si jednu z nich (v závislosti na svém identifikátoru) a tu dále rekurzivně dělí. Nevýhoda tohoto postupu je zřejmá, všechny procesory počítají až do úrovně vnoření *log k* totéž a pouze jeden z nich vždy výsledek ukládá, ostatní své výsledky zahazují.

Výsledky jak v případě přirozeného způsobu paralelizace, tak i v případě popsaného způsobu jsou podle [Cig93] překvapivě podobné. Testovaná paralelní implementace konstrukce Delaunayovy tetrahedronové sítě dosáhla pro uniformní data o 8000 bodech urychlení 1.7 – 3.35 pro 2 – 16 PEs na počítači nCUBE 2 system model 6410. Takto nízké urychlení je dáno nerovnoměrným zatížením procesorů. Nespornou výhodou algoritmu však je jeho použitelnost jak pro 2D tak i pro 3D.

```
Function ParDeWall(pid, callid: integer;
                   P: množina_bodů, AFL: seznam_stěn) : Triangulace
  var       f, fn : stěna; AFL^α, AFL^+, AFL^- : seznam_stěn;
            t : simplex; DT:  seznam_simplexů; α: dělící_rovina;
            P^+, P^- : množina_bodů;
  1. begin
  2.    AFL^α, AFL^+, AFL^- := []; DT := []; //v seznamech není žádná hrana
  3.    α := SelectPlane();              //dělící rovina se cyklicky volí
                                         //vždy rovnoběžně s x, y, z
        Pointset_Partition(P, α, P^+, P^-); //a dělí vstupní množinu na dvě
  4.    //spočti DeWall jak je uvedeno v algoritmu 4.2
        MakeWall(P, α, AFL, DT, AFL^α, AFL^+, AFL^-, P^+, P^-);
  5.    //rozhodni na základě pid zda výsledek zahodíme
        if not must_save(pid) then DT := [];
  6.    //aplikuj rekurzivně ParDeWall na tu z množin, kterou máme
        //spočítat v další úrovni volání (callid)
        if AFL^- <> [] and on_path(pid, callid*2) then
           DT := DT ∪ ParDeWall(pid, callid*2, P^-,AFL^-);
        if AFL^+ <> []and on_path(pid, callid*2 + 1) then
           DT := DT ∪ ParDeWall(pid, callid*2 + 1, P^+,AFL^+);
  7.    ParDeWall := DT;
  8. end; //end ParDeWall
```

Algoritmus 4.3: *Paralelizace DeWall algoritmu podle [Cig93].*

Realizační část

5 Paralelizace

Literatura neuvádí žádný paralelní algoritmus pro konstrukci Delaunayovy triangulace inkrementálním vkládáním dle schématu uvedeného v kapitole pojednávající o inkrementálním vkládání. Tato práce, která proto patří mezi průkopníky v této oblasti, si klade za cíl navrhnout paralelní algoritmus, který splní následující požadavky:

1. Velmi jednoduchá implementace
2. Dobré urychlení vůči sériové verzi pro všechny typy vstupních dat
3. Nevyžaduje speciální paralelní architekturu

Třetí požadavek splňují počítačové sítě a paralelní architektury se sdílenou pamětí. Využití počítačové sítě a protokolu *MPI* (Message Passing Interface) pro komunikaci mezi počítači si lze vzhledem k sekvenčnímu charakteru algoritmu představit jen stěží. Navíc komunikace jednotlivých procesů zasíláním zpráv nesplňuje požadavek velmi jednoduché implementace. Pro algoritmus, který má splnit oba požadavky, je k dispozici pouze architektura se sdílenou pamětí.

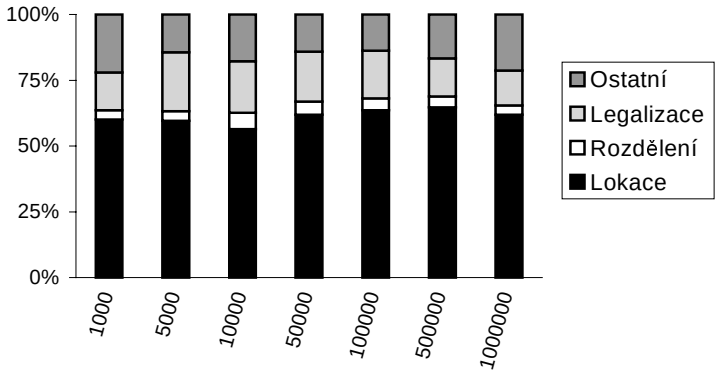
Předpokládejme tedy architekturu se sdílenou pamětí. Výpočet bude rozdělen mezi vlákna, na každém procesoru poběží jedno vlákno. Protože vlákna pracují se sdílenou *DAG* strukturou, bude nutné implementovat synchronizační mechanismus. Vysvětlení, oč se jedná, prozatím ponechejme a nyní proveďme analýzu sériového běhu algoritmu.

5.1 Analýza sériového algoritmu pro konstrukci *2D DT*

Jak je patrné ze zjednodušené verze algoritmu A.2, výsledný čas běhu algoritmu je dán součtem časů potřebných pro uskutečnění jednotlivých operací. Jedná se o:

1. čas potřebný pro lokaci, tj. čas potřebný pro vyhledání trojúhelníku, jenž obsahuje vkládaný bod, v *DAG* struktuře
2. čas potřebný pro rozdělení trojúhelníku na tři nové anebo rozdělení dvou trojúhelníků na čtyři nové
3. čas potřebný pro legalizaci vzniklé struktury
4. čas spotřebovaný ostatními operacemi algoritmu (např. randomizace množiny vstupních bodů, vytvoření dočasného počátečního trojúhelníku, uvolnění paměti alokované pro *DAG* strukturu atd.).

Zastoupení těchto dílčích časů ve výsledném času běhu algoritmu je uvedeno v grafu 5.1. Z něj vyplývá, že časově nejnáročnější operací je vyhledání trojúhelníku v *DAG* struktuře. Tato operace zabírá asi 60 – 70% celkového času potřebného pro konstrukci *DT*. Legalizace spotřebuje dalších 15 – 25%, rozdělení pouhých 3 – 5% a ostatní ryze sekvenční části algoritmu asi 15 – 25% času. Uzly *DAG* jsou alokovány i dealokovány postupně. Tento způsob je sice nejjednodušší, ale právě uvolňování paměti přidělené pro *DAG* strukturu tvoří značný podíl v kategorii ostatních operací. Za cenu plýtvání pamětí lze algoritmus velmi snadno upravit tak, že paměť pro uzly je alokována po souvislých blocích. Z experimentů vyplývá, že pro nesingulární typ dat je počet uzlů v *DAG* roven 6ti až 10ti násobku (pro singulární případy i 80ti násobku) počtu vstupních bodů. Pokud alokujeme paměť po blocích o velikosti šestinásobku počtu vstupních bodů, klesne časový podíl sekvenční části algoritmu pod 10%.

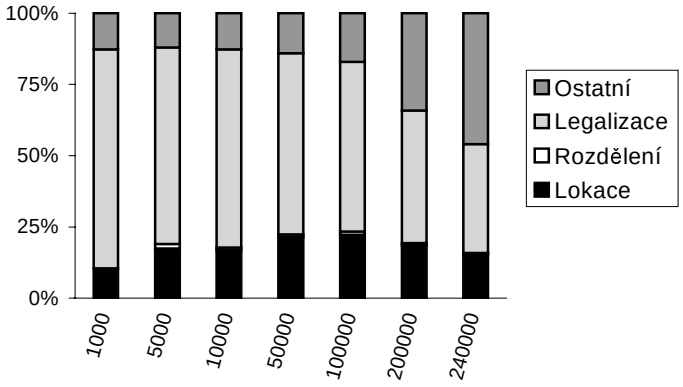


Graf 5.1: *Časový podíl hlavních částí základního 2D inkrementálního algoritmu pro uniformní data*

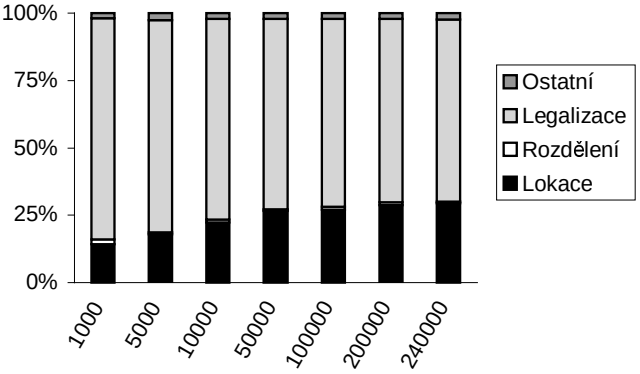
5.2 Analýza sériového algoritmu pro konstrukci *3D DT*

Krása algoritmu pro konstrukci *DT* inkrementálním vkládáním spočívá rovněž v jeho univerzalitě. Třírozměrná varianta algoritmu (viz algoritmus A.3) se od své dvourozměrné varianty liší pouze v poněkud jiném zastoupení časů jednotlivých operací ve výsledném čase běhu algoritmu. Uvažujeme-li postupnou alokaci i dealokaci uzlů, pak časově nejnáročnější operací (viz graf 5.2a) je tentokrát legalizace, která zaujímá asi 40 – 80% celkového času potřebného pro konstrukci *DT*. Lokace trvá okolo 10 – 40%, rozdělení necelé 2% a ostatní ryze sekvenční části algoritmu asi 8 – 60% času.

a) původní časový podíl jednotlivých fází algoritmu



b) časový podíl jednotlivých fází modifikovaného algoritmu

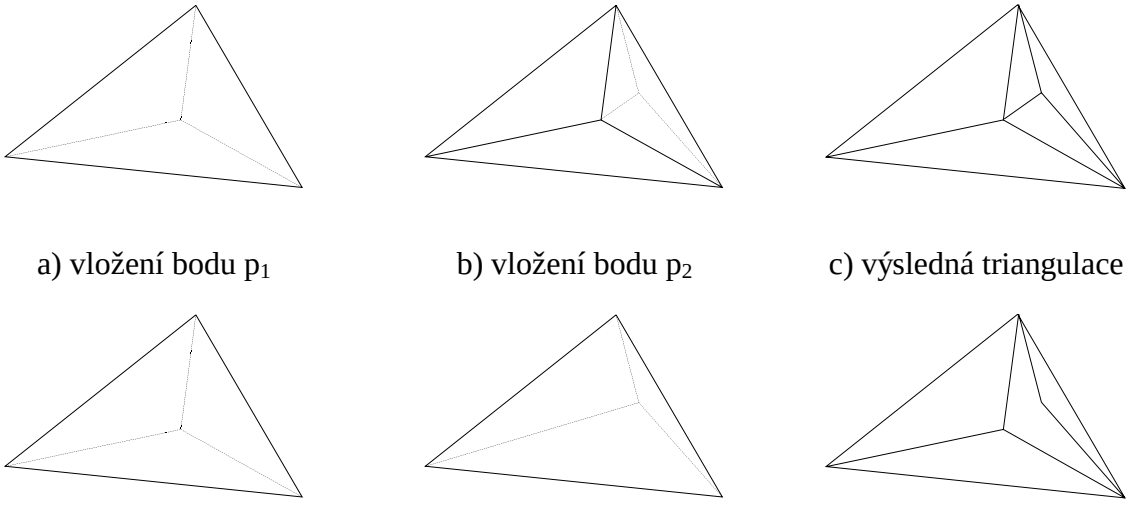


Graf 5.2: *Časový podíl hlavních částí základního 3D inkrementálního algoritmu pro uniformní data před úpravou (a) a po úpravě (b).*

Opět značný podíl, zejména pro větší počet vstupních bodů, přísluší uvolňování paměti přidělené *DAG* struktuře. Počet uzlů ve struktuře je úměrný 20ti až 60ti násobku *N*, kde *N* je počet vstupních bodů. Modifikací algoritmu tak, že uzly se alokují po blocích o velikosti 20N, se časový podíl drastickým způsobem změní (na rozdíl od 2D verze, kde modifikace nepřinesla příliš velkou úsporu). Stále nejnáročnější operací je legalizace a zaujímá asi 65 – 80% celkového času potřebného pro konstrukci *DT*. Lokace trvá okolo 15 – 30%, rozdělení necelé 2% a ostatní ryze sekvenční části algoritmu jen asi 3% času (viz graf 5.2b).

5.3 Synchronizační mechanismus

Pokud bychom použili sériový algoritmus beze změny a nechali pracovat všechna vlákna nezávisle, dříve nebo později dojde k vygenerování neplatného simplexu, resp. program havaruje. Problém působí fakt, že vlákna běží vůči sobě neodhadnutelnou rychlostí a pracují se společnými daty. Uvažujme dva body p_1 a p_2 ležící ve stejném simplexu. Dále předpokládejme činnost dvou vláken, jedno vkládá bod p_1 a druhé p_2. Obě vlákna alokovala paměť pro syny a přepsala ve stromové struktuře ukazatele na syny děleného uzlu. Pokud jsou tyto body p_1 a p_2 vloženy postupně, vznikne triangulace (viz obr. 5.1 a, b, c). Protože však vlákna běží různě rychle, může být nastavení ukazatelů prvního vlákna přepsáno anebo dokonce část nastavení může pocházet od jednoho vlákna a část od druhého. V obou případech *DAG* popisuje neplatnou strukturu (viz obr. 5.1 d, e, f).



a) vložení bodu p_1 b) vložení bodu p_2 c) výsledná triangulace

d) vložení bodu p_1 vláknem v_1 e) vložení bodu p_2 vláknem v_2 f) výsledná teselace[7]

Obr. 5.1: *Srovnání postupného (a, b, c) vkládání bodů p_1 a p_2 a souběžného (d, e, f). Na obrázku f je znázorněn pouze jeden z možných neplatných výsledků. Poznámka: legalizace není uvažována.*

Proto je nezbytně nutné zajistit, aby modifikace téhož uzlu neprovádělo více vláken současně. Běh vlákna musí být pozastaven a vlákno nemůže pokračovat, dokud konkurenční vlákno nedokončilo zápis. Mluvíme o synchronizaci činnosti vláken. V souvislosti se synchronizací se objevují termíny kritická sekce, mutex, semafor nebo událost [Jež99].

Jak sériový algoritmus pro dvourozměrný případ (viz A.2) , tak i jeho trojrozměrná varianta (viz A.3) pracuje se sdílenou strukturou *DAG* tak, že nejprve ji pouze čte – fáze lokace, a

[7] Teselací se nazývá dělení prostoru (roviny) do libovolných oblastí, ať již pravidelných či nepravidelných.

poté ji jak čte, tak i zapisuje – fáze rozdělení a legalizace. Protože čtení sdílených dat nepředstavuje žádný problém, lze fázi lokaci uskutečnit současně více vlákny, zatímco další fáze je nutno „ohlídat“. Existují tři základní náhledy na celou problematiku paralelizace uvedeného sériového algoritmu konstrukce *DT*: dávkový (batch), pesimistický (pessimistic) a optimistický (optimistic) přístup.

5.4 Dávkový přístup (Batch method)

Základní myšlenka dávkového přístupu spočívá v existenci dvou různých algoritmů, podle kterých běží jednotlivá vlákna.

Předpokládejme několik vláken, která budou současně vyhledávat v *DAG* struktuře trojúhelník obsahující vstupní bod, a jedno speciální vlákno, které bude provádět veškeré operace spojené s vlastním vložením bodu, tj. dohledání trojúhelníku, pokud vstupní trojúhelník byl v předchozí iteraci rozdělen, jeho rozdělení a legalizaci. Vyhledávací vlákno vloží dvojici vstupní bod a vstupní trojúhelník na konec fronty ve sdílené paměti, odkud specializované vlákno si dvojici vybere a zpracuje. Pokud je fronta prázdná, musí specializované vlákno čekat, pokud naopak je plná, musí počkat vyhledávací vlákno.

Z časové analýzy běhu sériového algoritmu v 2D vyplývá, že cca 75% času spotřebuje lokace, zbylých 25% pak rozdělení a legalizace. Jednoduchým srovnáním dospějeme k závěru, že již pro čtyři vyhledávací vlákna specializované vlákno nestíhá obsloužit požadavky a délka fronty narůstá až do jejího zahlcení. Jakmile dojde k zahlcení fronty, pravděpodobně dále poběží vždy jen dvě vlákna současně, ostatní budou čekat, až se pro jejich požadavek uvolní místo ve frontě. Navíc nejspíš naroste i počet uzlů, které musí specializované vlákno ještě otestovat, než nalezne trojúhelník pro rozdělení. Vzhledem k uvedeným problémům se nedá očekávat přijatelné urychlení pro více než 2 procesory.

Dávkový přístup si sice lze představit i v souvislosti s konstrukcí tetrahedronové sítě, ale nemá žádné opodstatnění. V tomto případě totiž předpokládáme jedno specializované vlákno pro vyhledávání a několik pro rozdělení a legalizaci (viz časová analýza z grafu 5.2). Pochopitelně je nutné synchronizovat činnost vkládacích vláken, neboť modifikují sdílené datové struktury. Navíc vzhledem k časovému poměru 3:7 lokace ku rozdělení a legalizaci zřejmě přijatelné urychlení nelze čekávat pro více než 2 procesory.

Vzhledem k těmto teoretickým výsledkům nebyl dávkový přístup programově realizován. Poznamenejme však, že pro jinou konkrétní sériovou implementaci s výhodnějším časovým poměrem obou fází může být tento přístup vhodným kandidátem.

5.5 Pesimistický přístup (Pessimistic method)

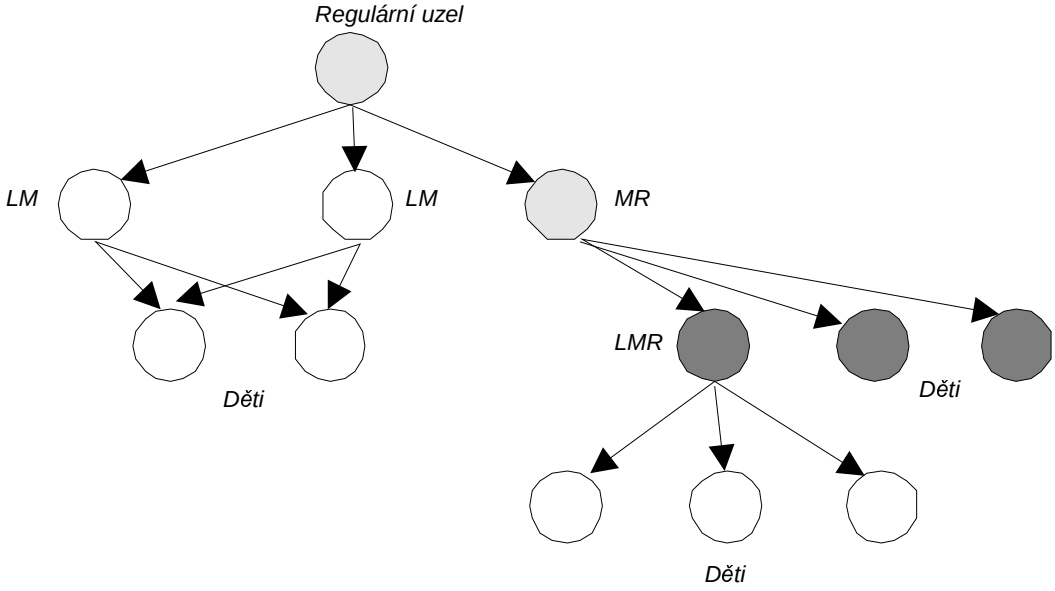
Pesimistický přístup vychází z dávkového zpracování. Hlavní změnou je neexistence specializovaného vlákna, všechna vlákna běží podle stejného programu. Zatímco lokaci mohou provádět všechna vlákna současně, pouze jedno vlákno smí modifikovat *DAG* strukturu. Proto vlákna naleznou poslední rodičovský trojúhelník, který obsahuje vstupní bod, poté vstoupí do kritické sekce, dokončí lokaci, rozdělí trojúhelník a provedou legalizaci, a nakonec z kritické sekce vystoupí.

Každý uzel v *DAG* struktuře obsahuje příznak kategorizující uzel (viz obr. 5.2). Rozlišujeme tři základní typy uzlů: děti, regulární uzly a poslední rodič. Listy v *DAG* jsou nazývány dětmi. Regulární uzel je takový uzel, který není listem a současně ani jeden z jeho synů není listem. Ostatní uzly jsou označené libovolnou kombinací písmen L, M a R. Pokud se v kombinaci vyskytuje písmeno L, znamená to, že levý syn uzlu je listem, tj. uzel nazván posledním L – rodičem. Obdobně výskyt M značí, že prostřední syn je listem (uzel je posledním M – rodičem) a pro R je listem pravý syn uzlu (poslední R – rodič).

Mějme vnitřní uzel *DAG* obsahující popis trojúhelníku, ve kterém se nachází vkládaný bod. Tento bod se rovněž nachází v trojúhelníku jednoho ze synů. Musíme otestovat jejich souřadnice, abychom zjistili, který z nich to je. Avšak je-li tento syn listem, nemůžeme vyloučit, že s ním jiné vlákno zrovna nepracuje. Proto smíme otestovat jen ty syny, které listy nejsou. Například v uzlu MR lze otestovat trojúhelník v levém synu, testování trojúhelníků v prostředním nebo pravém uzlu je přípustné až po vstupu do kritické sekce, a samozřejmě je uskutečněno pouze pokud vstupní bod neleží uvnitř trojúhelníku levého syna.

Rozšíření pro trojrozměrný případ je snadné. Protože tetrahedron se rozdělí na nejvýše čtyři nové, namísto M dostaneme dvojici M_L a M_R, a to je také jediná nutná změna.

Je zřejmé, že pesimistický přístup je jednoduchý, ale díky kritické sekci přináší jen omezené urychlení (viz kapitola Experimenty a výsledky).



Obr. 5.2: *Klasifikace uzlů DAG struktury. Písmena L, M a R u uzlů znamenají, že tento uzel nemá vnuka v levém (L), prostředním (M) či pravém (R) směru.*

Pesimistická verze pro konstrukci Delaunayovy triangulace inkrementálním vkládáním sestává z dvou dílčích algoritmů, a to algoritmu 5.1 pro hlavní vlákno a algoritmu 5.2 pro pracovní vlákna.

```
Master thread:
Input:   Množina S = {p_i, i = 0,1,...,N-1} bodů v rovině či prostoru
Output: Delaunayova triangulace DT(S)

1. begin
2.   Inicializuj počáteční velký simplex;
3.   Stanov náhodnou permutaci p_0, p_1, ..., p_n-1;
4.   Rozděl P na m podmnožin, kde m je počet vláken;
5.   Spusť všechna pracovní vlákna a čekej neaktivně dokud neskončí;
6.   Odstraň všechny simplexy obsahující vrcholy počátečního simplexu;
7. end;
```

Algoritmus 5.1: *Algoritmus hlavního vlákna pesimistické metody.*

```
Worker thread:
Input: Množina S_t = {p_i, i = 0,1,...,N_k-1} bodů v rovině či prostoru, S_t ⊂ S
Output:  Modifikuje sdílenou DT(S)

1. begin
2.   for r := 0 to N_k-1 do begin   // vlož p_r do DT(S)
3.     Zahaj lokaci trojúhelníku obsahující p_r až do úrovně rodičů;
4.     if libovolné vlákno pracuje s listy then čekej;
5.     Vstup do kritické sekce; //zahaj práci s listy
6.     Dokonči lokaci na úroveň listů a nalezni simplex
          obsahující p_r;
7.     Proveď rozdělení a zkontroluj nové simplexy;
8.     Vystup z kritické sekce; // ukonči práci s listy
9.   end;
10. end;
```

Algoritmus 5.2: *Algoritmus pracovního vlákna pesimistické metody.*

Hlavní vlákno aplikace inicializuje výpočet a spustí požadovaný počet pracovních vláken. Každé z těchto vláken vkládá do triangulace body ze své množiny vstupních bodů. Problému, jak optimálně rozdělit množinu vstupních bodů, se věnuje jedna z následujících samostatných podkapitol. Po spuštění všech pracovních vláken čeká hlavní vlákno neaktivně (tj. nezabírá procesorový čas), než jejich činnost skončí. Extrakci výsledné triangulace z *DAG* struktury a její případné uložení na disk, stejně jako uvolnění pomocné alokované paměti, provede sekvenčně hlavní vlákno. Paralelizace těchto úkolů je nesnadná, a, vzhledem k velikosti času potřebnému pro jejich uskutečnění, také zbytečná.

5.6 Optimistický přístup (Optimistic method)

Ačkoliv legalizace může změnit celou triangulaci, většinou jsou změny lokální, což znamená, že vlákno zpracuje jen několik málo listů. Pokud množinu vstupních bodů mezi dvě vlákna rozdělíme s respektováním polohy bodů (do dvou souvislých disjunktních množin) a vláknům povolíme provést kód z kritické sekce současně, zjistíme, že počet případů, kdy si vlákna „překážejí“, je minimální. Pesimistický přístup, který očekává výskyt problematického případu při vkládání každého bodu, se proto zdá být příliš pesimistický. Optimistický přístup vychází z opačného předpokladu, tj., že při vkládání bodu k problémům nedojde. Pro případ, že tento předpoklad není naplněn, je připravené řešení (může být komplikované a časově náročné), jak vzniklou situaci vyřešit.

Nechme všechna vlákna provádět jak lokaci, tak i rozdělení a legalizaci. Pochopitelně musíme zajistit synchronizaci mezi vlákny, tentokrát na jiné úrovni než při pesimistickém přístupu. Lokace je stejná jako při pesimistickém přístupu, to znamená „nechráněné“ prohledávání až na úroveň posledních rodičů. Do datové struktury uzlu *DAG* přidejme navíc příznak určující, zda je simplex uzamčen. Pracovní vlákno uzamkne ve fázi rozdělení a legalizace všechny simplexy, se kterými chce pracovat (i když je potřebuje jenom pro čtení), a jakmile je vložení bodu dokončeno, všechny simplexy jím uzamčené se odemknou. Pokud jiné vlákno již simplex uzamklo, vlákno, které tento simplex rovněž potřebuje ke své činnosti, musí počkat, dokud blokující vlákno neskončí svou činnost a neodemkne tento simplex. Optimistický přístup detailněji zachycuje algoritmus 5.3.

```
Worker thread:
Input: Množina S_t = {p_i, i = 0,1,...,N_k-1} bodů v rovině či prostoru, S_t ⊂ S
Output:  Modifikuje sdílenou DT(S)

1. begin
2.   for r := 0 to N_k-1 do begin   // vlož p_r do DT(S)
3.         Nalezni v DAG simplex obsahující p_r;
4.         while dělený simplex nebo jeho sousedé jsou uzamčeni
               někým jiným do čekej;
5.         Uzamkni simplex a všechny jeho sousedy;
6.         Rozděl simplex a uzamkni nové simplexy;
7.         while legalizované simplexy nebo jejich sousedé jsou
               uzamčeni někým jiným do čekej;
8.         Uzamkni legalizované simplexy a jejich sousedy;
9.         Legalizace;
10.        Odemkni všechny simplexy uzamčené tímto vláknem;
11.        Vzbuď všechna vlákna, která čekala na dokončení tohoto vlákna;
12.  end;
13. end;
```

Algoritmus 5.3: *Algoritmus pracovního vlákna optimistické metody.*

5.6.1 Problém uváznutí – deadlock

Předpokládejme dvě vlákna označená T_0 a T_1. Vlákno T_0 k dokončení své činnosti potřebuje simplex S_1, který je však uzamčen vláknem T_1. Proto vlákno T_0 musí počkat. Ale vlákno T_1 náhle zjistilo, že ke své činnosti potřebuje simplex S_0, který uzamklo vlákno T_0. Vlákno proto musí počkat, a tak dojde k uváznutí běhu aplikace.

Problém uváznutí lze vyřešit dvojím způsobem, a to buď zabránit jeho vzniku, nebo jeho vznik detekovat a nalézt optimální řešení.

Nechť každému vláknu je přidělena různá dynamická priorita. Na počátku se jedná o hodnoty 1 – *k*, kde *k* je počet vláken. Pokud vlákno vyžaduje přístup k simplexu, který uzamklo vlákno s vyšší dynamickou prioritou, vzdá se další činnosti, zruší změny jím provedené, odemkne všechny simplexy a počká na dokončení činnosti blokujícího vlákna. Poté zvýší svou dynamickou prioritu o *k* a provede opětovně dohledání, rozdělení a legalizaci za účelem konečně vložit svůj bod do *DT*. Tímto způsobem se zabrání vzniku uváznutí, metodu nazveme *optimistickou s prevencí deadlocku*.

Detekce uváznutí je snadná a spočívá v existenci seznamu „kdo na koho čeká“. Předpokládejme (pro příklad), že vlákno T_1 potřebuje simplex uzamčený vláknem T_2. Vlákno T_2 čeká na T_0, které je však aktivní, tj. na nikoho nečeká. Vlákno T_1 se přidá do seznamu a zahájí čekání. T_0 ale nyní potřebuje simplex uzamčený T_1. Vlákno T_1 čeká na T_2 a to čeká na T_0. Deadlock byl detekován a nyní je třeba nalézt řešení.

Jediné korektní řešení představuje, že buď vlákno, které deadlock detekovalo (v našem případě T_0) anebo vlákno, které deadlock zapříčinilo (v našem případě T_2), se vzdá další činnosti a „stáhne se“, přičemž zruší změny jím provedené, odemkne všechny simplexy a počká na dokončení činnosti blokujícího vlákna. Poté provede opětovně dohledání, rozdělení a legalizaci za účelem konečně vložit svůj bod do *DT*. Z našich experimentů vyplývá, že výhodnější je pokud se stáhne vlákno, které deadlock detekovalo. Tuto metodu nazveme *optimistickou s detekcí deadlocku*.

Oba přístupy k uváznutí (optimistická metoda s prevencí a optimistická metoda s detekcí) vedou k používání transakcí. Při každé modifikaci *DAG* struktury se dvojice – adresa měně-

ného obsahu paměti a původní hodnota – uloží do žurnálu. V případě, že vlákno je nuceno se stáhnout, provede se tzv. rollback znamenající, že provedené změny jsou na základě údajů v žurnálu stornovány a obsah žurnálu vymazán. Pokud vložení bodu je úspěšné, provede vlákno tzv. commit, což představuje pouhé vymazání žurnálu. Transakční mechanismus pochopitelně zvyšuje režii, a proto patří mezi limitující faktory urychlení.

5.6.2 Zlodějská metoda (burglary method)

Poměr počtu bodů, při jejichž vkládání došlo k nutnosti se stáhnout, ku počtu všech představuje setiny procenta (viz kapitola Experimenty a výsledky). V naprosté většině případů je tedy transakční mechanismus zbytečným luxusem. Zlodějská metoda je klonem optimistické s detekcí. Eliminuje transakční mechanismus za cenu možné existence ne-Delaunayovských simplexů v triangulaci. Protože však algoritmus automaticky detekuje potenciální problematický simplex, je možné ověřit jeho platnost v přídavné sekvenční fázi. Kromě eliminace transakcí metoda částečně odstraňuje nutnost zamykání simplexů.

Algoritmus zlodějské metody je analogií lidského života. Každá věc (např. televize, křeslo, židle, postel apod.) někomu patří a je umístěna v nějakém domě, který má rovněž svého vlastníka. Předpokládáme, že všechny věci v domě patří tomu, čí je dům. Je-li majitel, nazvěme ho Bobem, doma a sám, může si se svými věcmi dělat, co chce. Čas od času se stane, že Bob potřebuje věc, kterou nevlastní. Např. na opravu kalkulačky potřebuje pájku, ale tu má soused, nazvěme ho Billem. Nezbývá nic jiného, než se obléci a souseda navštívit. Bob zazvoní a počká, až mu Bill otevře. Dokud je Bob na návštěvě, ani jeden z nich nemá sám pro sebe všechny věci. V jednu chvíli Bob drží budík a Bill musí počkat až ho položí, pak zase Bill má v ruce vrtačku a Bob musí počkat. Chce-li ale Bill mít jak budík tak i vrtačku, nezbývá mu než Bobovi budík ukradnout – odtud název metody.

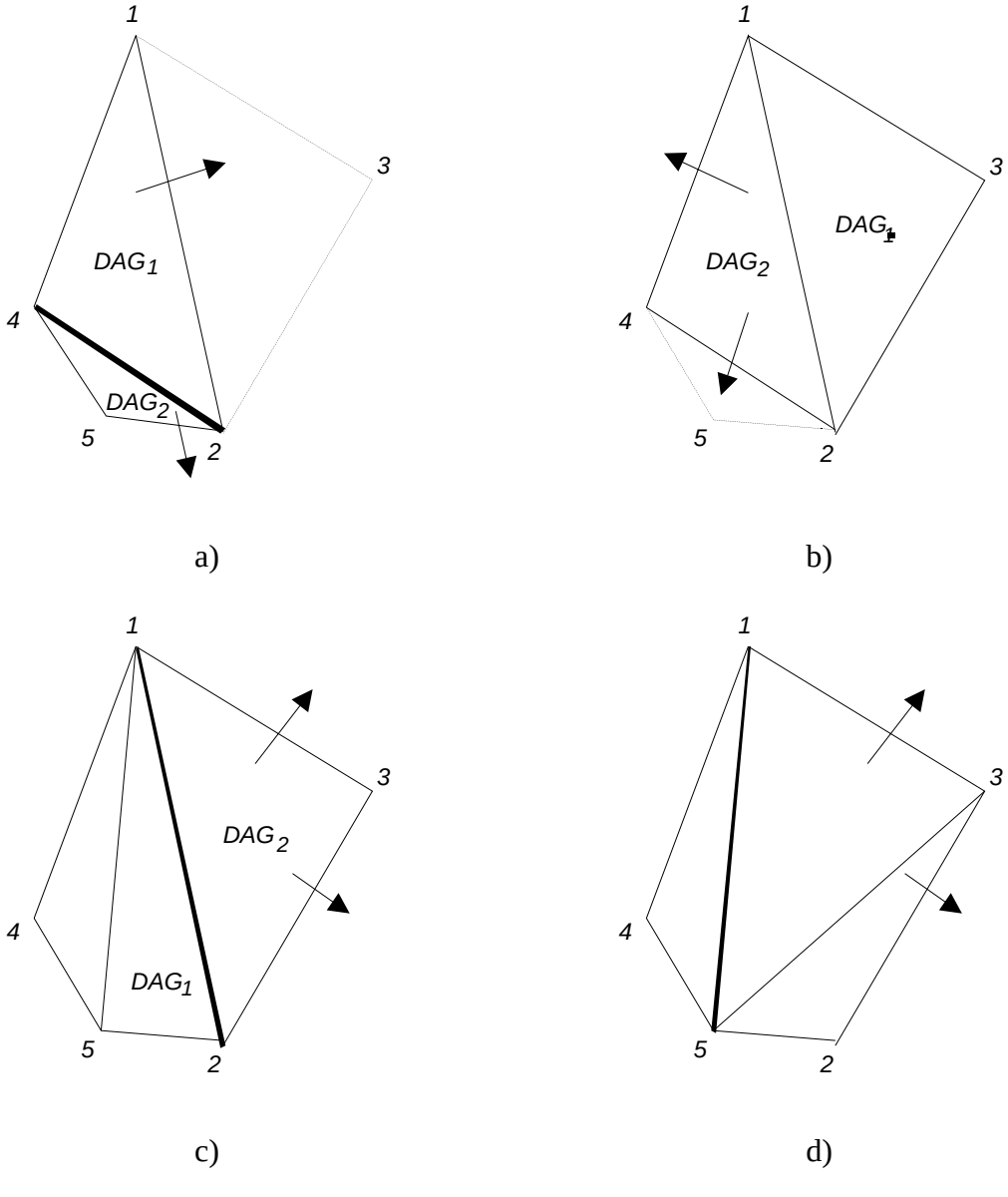
Rovina či prostor obsahující vstupní body se rozdělí na k částí (domů) obdobně jako v paralelním Bowyer-Watsonově algoritmu [Chr99]. Pokud vlákno (vlastník) pracuje se svými simplexy (věci) a je sám, neuzamyká. Pokud vlákno pracuje na cizím území anebo na jeho území se nachází alespoň jedno cizí vlákno, musí uzamykat. Na rozdíl od standardní optimistické metody, vlákno smí používat i simplexy, které neuzamklo pro sebe, pokud vlastní příslušný „klíč“. Klíč je vláknu přidělen v okamžiku, kdy detekovalo deadlock. Pokud např. vlákno T_0 detekovalo deadlock, ke kterému by došlo, kdyby zahájilo čekání na T_1, dostane nově klíč K_1. Nadále bude moci používat všechny simplexy uzamčené buď jím samým anebo vláknem T_1. Propůjčený klíč vlákno ztratí, jakmile dokončí vložení svého bodu a odemkne simplexy jím uzamčené. Je zřejmé, že protože T_1 čeká, nedojde k souběhu více vláken vedoucímu k obecné teselaci. Otázkou zůstává, zda vzniklá triangulace je Delaunayova.

Vnikl-li vetřelec na území vlákna, které právě uzamykalo sousedy, aby mohlo provést rozdělení, triangulace bude Delaunayovská. Přerušené vlákno totiž dosud neprovedlo jedinou modifikaci a tedy je možné, aby se stáhlo, odemklo své trojúhelníky, opětovně dohledalo patřičný trojúhelník a operaci dokončilo.

Předpokládejme však, že před přerušením činnosti T_0 detekovalo ne-Delaunayovský trojúhelník v DAG (nazvěme ho DAG_1) a jeho hrana sdílená se sousedním trojúhelníkem (nazvěme ho DAG_2) se musí prohodit. Požadavek na uzamčení DAG_2 musel být odložen, protože tento simplex vlastnilo vlákno T_1, provádějící též legalizaci (jinak by ke konfliktu nedošlo). T_0 proto zahájilo čekání na T_1. Nešťastnou shodou okolností vlákno T_1 v rámci své zlodějské činnosti rozdělilo DAG_1. Po obnově činnosti přerušeného vlákna, vlákno T_0 již nemůže normálně pokračovat, protože výchozí trojúhelník v podstatě neexistuje (byl rozdělen). Je sice pravda, že po modifikaci triangulace muselo zlodějské vlákno legalizovat nově vytvořené trojúhelníky, ale jen ve směrech, kde legalizace ještě nebyla provedena, proto bohužel výsledek není vždy DT.

Obrázek 5.3 znázorňuje popsaný problém. Šipkami jsou znázorněné hrany, které budou po prohození tučně označené hrany ověřeny, zda jsou legální. Vlákno T_0 detekovalo neplatnou hranu a hodlá ji zaměnit. K dokončení operace však potřebuje uzamknout sousedy jak DAG_1, tak i DAG_2 (viz 5.3a). Trojúhelník *4*, *5*, *2* ale momentálně není volný, T_0 musí počkat, než vlákno T_1 simplex uvolní. Toto vlákno ale zrovna chce otestovat platnost tohoto trojúhelníku a k tomu potřebuje přístup k trojúhelníku *1*, *4*, *2* (viz 5.3b). Protože čekání by způsobilo uváznutí, vlákno trojúhelník „ukradne“.

Vlákno T_1 detekovalo neplatnou hranu a provede záměnu. Nyní ověří hrany *2*, *5* a *1*, *2*. Druhá hrana je nelegální, provede další záměnu (viz 5.3c). Záměnou se nelegální hranou však stane hrana *1*, *5* (viz 5.3d). Vlákno T_1 legalizaci hrany neuskuteční, protože není ve směru postupující legalizační vlny. Přerušené vlákno legalizaci neuskuteční, protože jeho operaci nelze dokončit.



Obr. 5.3: *Konflikt dvou vláken ve zlodějské metodě vedoucí ke vzniku obecné triangulace místo DT. Tučně označeny neplatné hrany, šipkami směr šíření legalizace.*

Po dokončení činnosti zlodějského vlákna existuje stále jedna nelegální hrana. Pokud přerušené vlákno, které nemůže dokončit operaci, se vzdá další činnosti v této oblasti, zůstanou ve výsledné triangulaci nelegální hrany a nejedná se tedy o *DT*.

Existují tři možné přístupy jak se s problémem chybné triangulace vypořádat. Při prvním si algoritmus simplex *DAG* zapamatuje a po dokončení činnosti všech vláken se sekvenčně zkontroluje jeho okolí (včetně potomků). Druhý způsob se liší pouze v načasování verifikace – ověření se provede okamžitě. Vystává však otázka, jak ošetřit možný detekovaný deadlock druhé úrovně, protože před ověřením je samozřejmě nutné uzamknout potomky problémového simplexu, což může vést k uváznutí. Poslední přístup znamená, že se smíříme s triangulací, která není globálně Delaunayovská. Experimenty ukazují, že pravděpodobnost poruchy v síti je relativně malá, navíc může být ještě snížena vhodným rozdělením množiny vstupních bodů mezi procesory. Proto často výsledkem je opravdu *DT*. Kromě toho, ve zkoumaných chybových případech jsme nikdy nenalezli tak špatnou dvojici trojúhelníků, která by nebyla pro většinu aplikací přijatelná. Toto je patrné i na obrázku 5.3, kde poměr délky správné hrany ku délce hrany ponechané se blíží k jedné.

5.6.3 Metoda vícenásobného zamykání (advanced burglary method)

Bylo již naznačeno, že standardní optimistická metoda uzamčené simplexy odemyká až po dokončení vkládání. Je zřejmé, že čím větší jsou uzamčené prostory uzlů, s tím větší pravděpodobností dojde ke kolizi dvou vláken a tudíž k čekání a potenciálně i k deadlocku. Na druhou stranu, k operaci vlákno potřebuje pouze malý prostor uzlů, pravděpodobnost, že kolize nastane v místě, kde vlákno právě operuje, je malá. Tato metoda obdobně jako předchozí metoda eliminuje transakční mechanismus. Uzamyká sice neustále, ale uzly odemyká ihned, jakmile již nejsou potřeba. Rozhodnutí, zda uzel již může být odemknut, lze jednoduše provést zavedením lokálního statického pole, do kterého si operace uloží nově zamčené simplexy potřebné pro její vlastní činnost. Tyto simplexy, stejně jako simplexy na vrcholu standardního zásobníku uzamčených uzlů, budou odemčeny. Nově vzniklé simplexy se uloží na vrchol globálního zásobníku pro další legalizaci. Postup předpokládá pevnou strukturu globálního zásobníku. Očekává-li se na vrcholu uzamčený rodičovský uzel, musí tam také být. Do datové struktury uzlu přidáme další atribut, který zvýšíme o jedna vždy při žádosti uzel uzamknout a snížíme vždy při žádosti uzel odemknout. Pokud měníme hodnotu atributu z nuly na jedna, provedeme fyzické uzamčení uzlu, naopak přechod z jedné na nulu způsobí jeho odemčení.

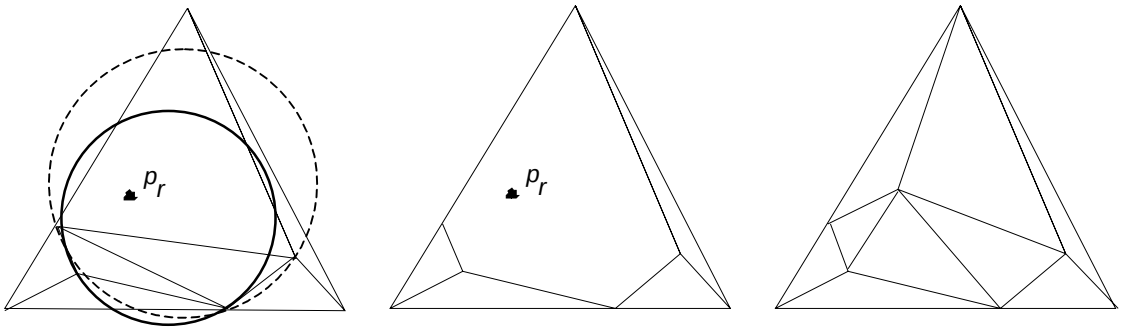
Třebaže předpokládáme snížení pravděpodobnosti vzniku deadlocku, nemůžeme ho zcela vyloučit. Transakcím se chceme vyhnout a navíc vzhledem k průběžnému odemykání ani nepřichází v úvahu. Rovněž zlodějský způsob popsaný v předchozím textu není vyhovující, protože díky průběžnému odemykání může být blokované vlákno aktivováno ještě před dokončením zlodějské činnosti druhého vlákna, a tak může dojít k souběhu.

Vyřešíme problém uváznutí proto takto: Detekuje-li vlákno deadlock, převezme uzel, což znamená změnit příznak uzamčení na sebe, nastavit počet zámků na jedna a označit uzel jako kradený. Odemyká-li (fyzicky) zloděj kradený uzel, vrátí příznak uzamčení na původního vlastníka, obnoví hodnotu počtu zámků a aktivuje blokované vlákno.

5.6.4 Metoda kružnice opsané (circle method)

Ačkoliv myšlenkově se jedná o jednu z nejstarších metod, vývojově je nejmladší. Vychází z principu Bowyer–Watsonova sekvenčního algoritmu pro konstrukci *2D DT*, který místo vložení bodu do trojúhelníku a jeho rozdělení ověří všechny trojúhelníky v jeho sousedství a odstraní z triangulace ty trojúhelníky, v jejichž kružnici opsané leží vkládaný bod. Tím vznikne díra, která se retriangularizuje prostým spojením původních bodů na hranici díry a nového vkládaného bodu. Činnost algoritmu je znázorněna na obrázku 5.4. Dva trojúhel-

níky porušují princip prázdné kružnice opsané a jsou proto odstraněny, díra je retriangularizována. Srovnejte výslednou triangulaci s triangulací na obrázku 3.6, která však byla dosažena lokálními transformacemi.



Obr. 5.4: *Konstrukce Delaunayovy triangulace vytvořením díry a její retriangularicí.*

Jinými slovy, legalizace se týká pouze těch trojúhelníků, v jejichž kružnici opsané se nachází vkládaný bod. Potřebuje-li vlákno T_0 ke své činnosti trojúhelník S_0 a v kružnici opsané tomuto trojúhelníku neleží žádný z bodů vkládaných ostatními vlákny, může s trojúhelníkem pracovat. Není-li podmínka splněna, například v kružnici se nachází bod vkládaný vláknem T_1, musí vlákno T_0 počkat.

Metoda je extrémně jednoduchá, na rozdíl od předchozích metod neuzamyká, na druhou stranu dojde-li ke konfliktu, jedná se vždy o uváznutí (deadlock)! To lze vyřešit obdobným způsobem jako ve zlodějské metodě. Poznamenejme, že výsledná triangulace může obsahovat defekty. Metoda je použitelná také v trojrozměrném případě, ale protože (narozdíl od 2D) koule opsaná může být příliš veliká a pokrývat značnou část prostoru, nezdá se být vhodnou.

5.7 Rozdělení množiny vstupních bodů mezi jednotlivá vlákna

Rozlišujeme dva základní principy, jak rozdělit množinu vstupních bodů na *k* podmnožin:

1. Množina je staticky rozdělena. Problémem však je, že poslednímu vláknu se přidělí méně než ostatním. Protože však předpokládáme architekturu pouze s několika procesory, rozdíl je nepatrný (v porovnání s velikostí úlohy). Experimenty vykazují minimální časový rozdíl mezi konci běhu nejrychlejšího a nejpomalejšího vlákna.
2. Každé vlákno vkládá první dosud nevložený bod, na který ukazuje sdílený ukazatel, jež se atomicky zvyšuje. Jedná se o dynamické dělení množiny.

Zvolili jsme první způsob, protože se jedná o jednodušší způsob a navíc rozdíl mezi velikostmi podmnožin je zanedbatelný v porovnání s velikostí podmnožiny.

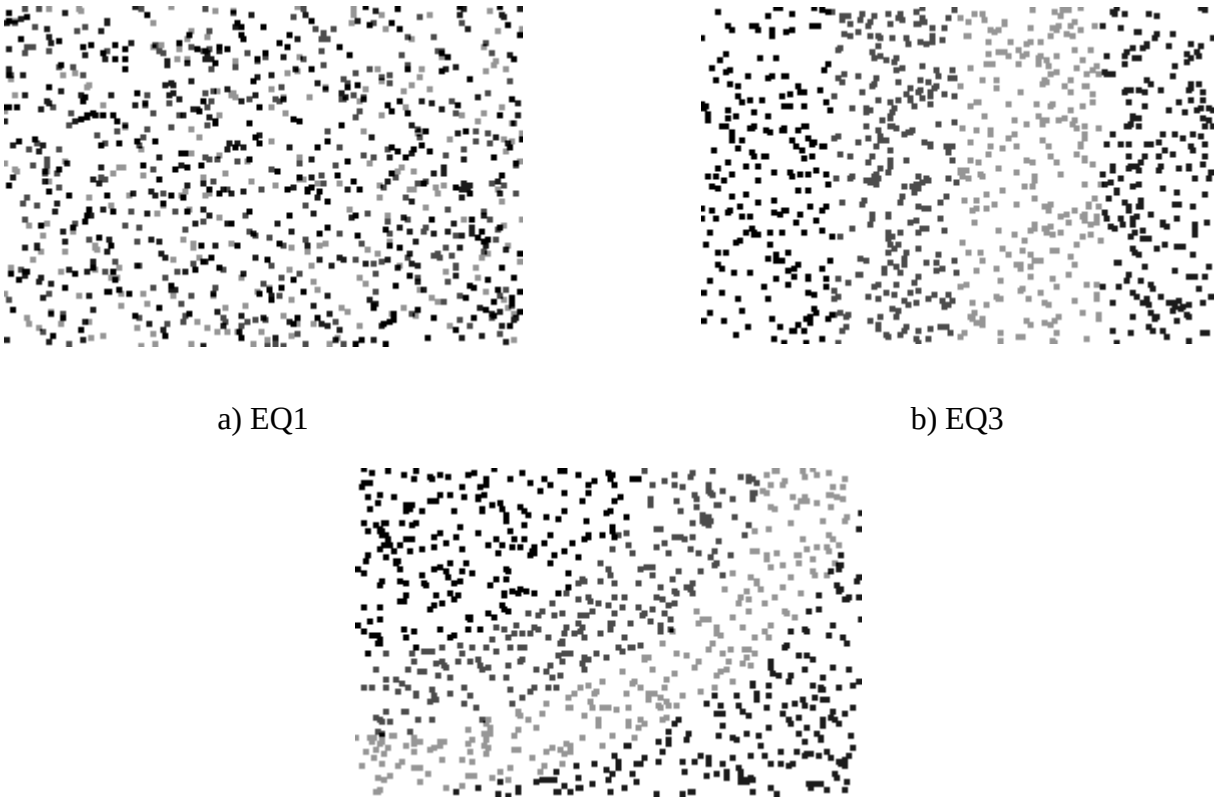
Jak již bylo napsáno dříve, je vhodné pro sériový algoritmus randomizovat pořadí vkládání bodů. Rozdělíme-li jednoduše[8] takto randomizovanou množinu mezi jednotlivá vlákna, je to nejlepší, co můžeme udělat? Tento způsob budeme označovat zkráceně *EQ1*. Obrázek 5.5a znázorňuje geometrickou polohu bodů přidělených vláknům. Dá se očekávat vyšší počet konfliktů a tudíž i vyšší čas potřebný na sestrojení *DT*. Vhodnější se zdá být rozdělení vstupních bodů v souladu s jejich geometrickou pozicí.

[8] Poznamenejme, že takovéto rozdělení je možné jen díky použité architektuře se sdílenou pamětí.

Každé vlákno obdrží body s *x*-souřadnicí v pásu $<x_i,x_{i+1}>$ kde *i* je index hranice pásu. Hranice se spočítá modifikovaným algoritmem pro nalezení přibližného mediánu[9], jehož složitost je v nejhorším případě *O(n)*, ale v očekávaném případě poskytuje lepší výsledek. Pochopitelně lze setřídit vstupní body podle jejich *x*-souřadnice, rozdělit je mezi jednotlivá vlákna a každé vlákno před zahájením vkládání si svou množinu randomizuje. Nejlepší sériová verze pro třídění má však složitost $O(n \cdot log(n))$ a je tedy mnohonásobně pomalejší než modifikovaný algoritmus pro medián. Literatura sice uvádí paralelní verzi QuickSortu, avšak ani tak „sort nestačí na medián“. Výsledek tohoto způsobu distribuce, označovaného zkráceně *EQ3*, uvádí obrázek 5.5b. Čím více vláken do výpočtu je zapojeno, tím užší pásy se vláknům přidělují. Lze proto uvažovat o vylepšení, které spočívá v zahrnutí druhé souřadnice. Např. pro čtyři vlákna nejprve nalezneme medián pro *x*-souřadnici a poté i pro *y*-souřadnici a prostor rozdělíme na čtyři obdélníkové oblasti.

Jiným a snadnějším způsobem, jak zahrnout do distribuce i *y*-souřadnici, spočívá v setřídění bodů dle jejich vzdálenosti od počátku (tj. $x^2 + y^2$). Výsledek tohoto způsobu distribuce, označovaného zkráceně *EQ5*, uvádí obrázek 5.5c.

Analogicky zahrneme také *z*-souřadnici. Poznamenejme, že výsledná distribuce označovaná jako *EQ7*, přichází v úvahu pouze pro trojrozměrný případ.



a) EQ1 b) EQ3

c) EQ5

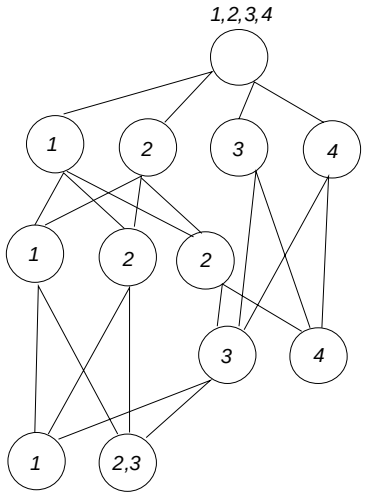
Obr. 5.5: *Metody distribuce množiny vstupních bodů mezi čtyři vlákna.*

[9] Implementaci poskytli ing. Šimána a ing. Kroc.

V případě takovýchto distribucí však musíme mírně upravit zlodějskou metodu tak, že použijeme virtuální „domy“. Když vlákno vytvoří nové simplexy, ať již na svém území (viz *EQ3* a *EQ5*) anebo na cizím, přivlastní si je. Samozřejmě při použití *EQ1* bude vláknem spravovaná část nekompaktní, ale *EQ3* i *EQ5* (resp. *EQ7*) by tento problém měly vyřešit, neboť očekáváme prolnutí jen na hranicích, zbylá část by měla zcela patřit jednomu vláknu.

Distribuce označované *EQ2*, *EQ4* a *EQ6* rovněž existují, ale jejich význam zanikl zakomponováním mediánu, a proto se o nich nebudeme nadále zmiňovat.

Způsob distribuce označovaný jako *EQB1* vychází z myšlenky paralelního Bowyer-Watsonova algoritmu [viz Chr99]. Prostor je dělen mezi jednotlivá vlákna dynamicky během výpočtu s cílem vygenerovat nejkratší hranici a stejný počet uzlů pro každé vlákno. Uzel může mít více vlastníků, při jeho rozdělení se vlastníci distribuují. Dojde-li k rozdělení uzlu na *k* nových (tj. 2, 3 a ve 3D ještě 4), jsou kombinováni vlastníci všech rodičů (fúze) a tito vlastníci rozděleni mezi nově vzniklé uzly tak, aby počet vlastníků s hloubkou stromu klesal a přitom buď žádný vlastník nebyl diskriminován, anebo, pokud není vyhnutí, byl diskriminován ten, který má aktuálně největší prostor. Současně „vzdálenost“ sousedních uzlů od nového musí být minimální (tj. kompaktnost). Výchozí simplex je vlastněn všemi vlákny, listy mají většinou jediného vlastníka. Odpovídající změnu v *DAG* dokumentuje obrázek 5.6.



Obr. 5.6: *Vlastnictví simplexů v triangulaci podle EBQ1 distribuce množiny vstupních bodů mezi čtyři vlákna (s identifikátory 1, 2, 3 a 4).*

6 Implementace

Pro zajištění přenositelnosti kódu by mělo být paralelní řešení konstrukce Delaunayovy triangulace implementováno v ANSI C. Nicméně již fakt, že každý operační systém poskytuje odlišné prostředky pro paralelizaci, snižuje přenositelnost kódu. Přednost dostaly platformy Microsoft Windows zejména proto, že jedním požadavkem této práce je začlenění do MVE a tento systém pracuje pouze na těchto platformách. Dalším argumentem je existence odladěné sériové verze algoritmu, která byla implementována v Borland Delphi 5.0 a jejíž autorkou je Doc. Dr. Ing. Ivana Kolingerová. Vzhledem k těmto okolnostem nejlepší volbou by bylo využít Borland Delphi a doplnit paralelizaci do již existujícího sekvenčního kódu. Autor práce však plně neovládá tento „programovací jazyk", preferuje výhradně C++ a rovněž vývojový nástroj Microsoft Visual C++ 6.0 mu poskytuje nejlepší prostředky pro vývoj jakékoliv aplikace. Algoritmus byl proto napsán a odladěn v prostředí Microsoft Visual C++ v 6.0, později v MS Visual Studio.NET 7.0 a to jak ve formě samostatné aplikace, tak i jako DLL knihovna do systému MVE (viz podkapitola Integrace do MVE). Pro zjednodušení komunikačního rozhraní uživatel – aplikace se využilo staticky linkované knihovny MFC[10].

Operační systém MS Windows poskytuje velké množství synchronizačních prostředků, jako jsou mutexy, semafory, události anebo zajištění kritické sekce. Chybějící relativně často užívanou bariéru lze snadno implementovat pomocí událostí. Problémem všech těchto synchronizačních prostředků je jejich režie. Uzamykání a odemykání simplexů se musí provést jako nedělitelná (atomická) operace; a tato operace se vykonává v 3D verzi průměrně 260krát pro každý vkládaný bod. Ověřili jsme experimentálně, že použití kritické sekce zpomalí dvourozměrnou verzi algoritmu natolik, že optimistický přístup přináší nižší urychlení než pesimistický. Naštěstí většinou, byť za cenu duplicitního testování, se lze kritické sekci vyhnout, protože existují funkce poskytované operačním systémem, které dokáží provést atomicky jednoduchou instrukci jako je zvýšení či snížení proměnné o jedničku[11], zvýšení či snížení proměnné o hodnotu jiné proměnné[12] či změna hodnoty proměnné za předpokladu rovnosti původní a zadané[13] hodnoty; a tyto funkce postačují. Srovnání efektivity prostředků přináší tabulka 6.1. Z ní je patrné, že zajištění triviálního kódu kritickou sekcí zpomalí běh algoritmu 10x, zatímco při použití atomických funkcí jen 5x.

Implementační podrobnosti, stejně tak i uživatelskou dokumentaci samostatné aplikace naleznete v příloze B. Uživatelská dokumentace k MVE modulu je uvedena v příloze C.

Nezajištěný kód	Čas	MFC kritická sekce	Čas
a++;	100	MFC_EnterCS; a++; MFC_LeaveCS;	1034
WINAPI kritická sekce	**Čas**	**Atomická funkce**	**Čas**
EnterCS; a++; LeaveCS;	1030	InterlockedIncrement(&a);	533

Tabulka 6.1: *Různé způsoby zajištění triviálního kódu a jim odpovídající potřebný počet časových jednotek.*

[10] Microsoft Foundation Class je knihovna zapouzdřující většinu WINAPI funkcí do objektového modelu.

[11] Intel x86 kompatibilní systémy disponují pro tento případ instrukcemi *lock INC* a *lock DEC*.

[12] Intel x86 kompatibilní systémy disponují pro tento případ instrukcí *XADD*.

[13] Intel x86 kompatibilní systémy disponují pro tento případ instrukcí *CMPXCHG*.

6.1 Integrace do MVE

MVE je zkratka pro Modular Visualization Environment [viz MVE], systém vyvíjený týmem počítačových grafiků (CGG) na Západočeské univerzitě v Plzni. Jak již název napovídá, jedná se o systém zahrnující knihovnu nejrůznějších modulů a jednoduchý editor, který umožňuje uživatelům interaktivní spojování a spouštění těchto modulů. Modulem se zde rozumí „obyčejný“ program, který produkuje nějakou činnost. Modul například generuje či nahrává data ze souboru, nebo je naopak ukládá, výpočetní moduly zpracovávají vstupní data (např. konstruují trojúhelníkové sítě nebo je zobrazují). V nejjednodušším případě pouhým pospojováním výstupů jednoho modulu se vstupy jiných a nastavením parametrů všech zúčastněných modulů vytvoří uživatel (i bez programovacích znalostí) „aplikaci“ řešící daný problém. Programátor nastupuje teprve, když je třeba implementovat nový algoritmus.

Systém programátorům poskytuje možnost jednoduchého zabudování vlastních částí mezi již existující celky. Navíc programátor nemusí psát tolik kódu jako v případě samostatné aplikace se stejnou funkčností. Např. plně postačuje mít jeden vizualizační modul, který zobrazuje výsledky ostatních výpočetních modulů (můžeme tak odstranit zbytečné programování podobných vizualizačních částí ve více různých aplikacích). Z programátorského hlediska je MVE řešeno jako systém dynamických knihoven pod operačním systémem Microsoft Windows. V jedné knihovně může být umístěno i několik modulů, přičemž každý modul je vytvořen pro svoji konkrétní činnost. Knihovny komunikují s editorem pomocí jednoduchého rozhraní několika jasně definovaných funkcí. Implementace algoritmu spočívá tedy ve vytvoření jednoho či více modulů, které jsou umístěny v jedné nebo více DLL knihovnách.

Implementované paralelní řešení pro konstrukci *DT* je obsaženo ve dvou modulech, jeden nazvaný *Parallel_DT* pro dvourozměrný případ a druhý nazvaný *Parallel_DT3D* pro případ trojrozměrný. Moduly lze kategorizovat jako výpočetní, jejich výstupem je trojúhelníková nebo tetrahedronová síť (modul *DT3Dauxlib* extrahuje z tetrahedronové sítě povrch a vrací ho v podobě trojúhelníkové sítě, která je zobrazitelná vizualizačním modulem). Množina vstupních bodů je generována nebo načtena ze souboru přímo modulem. Modul pro konstrukci 2D *DT* zapouzdřuje *Parallel_DTLib.dll*, která využívá *SDT.dll* knihovnu se sériovou verzí algoritmu. Obdobně modul pro trojrozměrnou variantu zapouzdřuje knihovna *Parallel_DT3DLib.dll*, která využívá *DT3DLib.dll* knihovnu se sériovou verzí algoritmu.

Popis činnosti z uživatelského hlediska je uveden v dokumentu, který je součástí dokumentace modulů systému MVE. Zkrácenou verzi naleznete v příloze C.

7 Experimenty a výsledky

Prezentované výsledky pocházejí z testů, které probíhaly na různých počítačích i platformách. Vývoji a základnímu testování posloužil osobní dvouprocesorový počítač 2x Intel Pentium Celeron 533MHz, 512MB RAM s operačním systémem Microsoft Windows 2000. Primární testy se uskutečnily na laboratorním počítači Západočeské Univerzity v Plzni – Dell Precision 410, jehož parametry jsou: 2x Pentium III, 500 MHz, 1 GB RAM, Microsoft Windows NT 4.0. Díky patří firmě Dell Computer, Czech Republic, která nám umožnila otestování aplikace na jejich multiprocesorových počítačích. Prezentace výsledků poskytnutých testováním na stroji Dell PowerEdge 7150 tvoří jádro této kapitoly. Dell PowerEdge 7150 je 64 bitový stroj (4x Intel Itanium, L3 cache 4MB, 800 MHz, 8GB ECC SDRAM s 256 Mega transferů za sekundu) s Microsoft Windows XP Advanced Server 64 bitovým operačním systémem. Pravděpodobně běh 32-bitové aplikace je nějak emulován, protože výsledné časy ve 3D jsou asi 5x vyšší než výsledné časy týchž měření provedených na osobním počítači. Ať již vinu nese emulace, lepší algoritmus pro vstup do kritické sekce v 64-bitovém jádře OS, 4MB L3 cache nebo něco jiného, faktem zůstává, že dosahovaná urychlení na tomto stroji bývají vyšší než urychlení pro stejná data na jiných strojích (pro dvě vlákna je urychlení pesimistické metody ve 3D o 4 – 7% vyšší než urychlení dosažené na 2x Intel Celeron 533MHz).

Chování navržených algoritmů na počítači s více procesory je rovněž prezentováno, ale tyto výsledky nejsou platné pro aktuální verzi aplikace (vzhledem k omezené možnosti přístupu k víceprocesorovým počítačům se nepodařilo dosud otestovat aktuální verzi). Tyto uvedené výsledky pochází z testování na stroji Dell PowerEdge 8450 (8x Intel Pentium III Xeon, cache 2MB, 550 MHz, 2GB RAM) s operačním systémem Microsoft Windows 2000 Server a na stroji společnosti UniSys (které rovněž patří naše poděkování) ES5000 (8x Intel Pentium III Xeon, cache 2MB, 700 MHz, 2GB RAM) s operačním systémem Microsoft Windows 2000 Datacenter.

Veškerá prezentovaná urychlení byla vypočtena jako celkový čas potřebný pro konstrukci *DT* sériovou metodou ku celkovému času potřebnému pro paralelní metodu. Operace načítání množiny z disku do paměti a ukládání triangulace na disk nejsou ve výpočtu uvažovány.

7.1 Vstupní data

Stabilita a efektivita algoritmů bývá často ovlivněna charakterem vstupních dat. V této podkapitole uvedeme zástupce nejvýznamnějších typů (včetně problematických singularit), které byly použity pro ověření navrženého algoritmu.

Jak již bylo řečeno v úvodu, množina vstupních bodů může mít podobu rozptýlených dat (nikterak strukturovaná množina) nebo tato data mohou být strukturována do podoby více či méně pravidelných mřížek. Mezi zástupce pravidelných dat, která byla testována, patří tzv. regulární, resp. „gridová“ data. Body v tomto rozložení jsou umístěny v pravidelném pravoúhlém rastru dělícím interval <0,1> na přibližně $\sqrt{N} x \sqrt{N}$ buněk, kde N je počet bodů. Body se generují s použitím rovnoměrného rozložení.

Skupinu rozptýlených datových rozložení lze rozdělit na podskupinu s rovnoměrným rozptýlením (tzv. uniformní data) a s rozptýlením nerovnoměrným (tzv. neuniformní data). Ukázky některých významnějších rozptýlených datových rozložení lze spatřit na obrázku 7.1.

Uniformní rozložení má vstupní body rozptýleny rovnoměrně ve čtvercové oblasti. Souřadnice x a y jsou generovány náhodně s rovnoměrným rozdělením v jednotkovém čtverci.

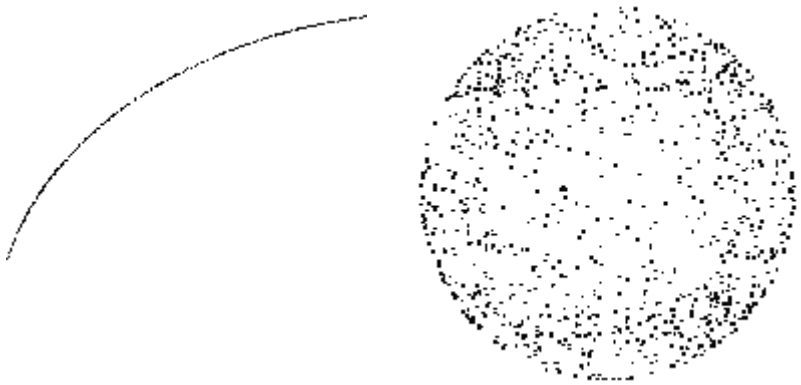
Normální (gaussovské) rozložení je typickým příkladem neuniformních dat. Souřadnice *x* a *y* jsou generovány náhodně s normálním rozdělením o parametrech μ, σ. Výsledkem jsou data uspořádána radiálně, hustota bodů ve středu je však mnohem větší než hustota na kraji.

„Cluster“ rozložení vznikne kombinací několika skupin normálních dat vygenerovaných ve stejné oblasti, ale s různým středem (hodnotou parametru μ). Tímto postupem vzniknou vstupní data, která tvoří v oblasti izolované shluky. Poznamenejme, že shluky mohou být vygenerovány rovněž v souladu s Kuzminovým rozložení [Ble96].

a) Uniformní distribuce b) Normální distribuce c) Shluky

Obr. 7.1: *Příklad různých rozptýlených distribucí ve 2D.*

Mezi singulární případy patří rozložení bodů na přímce. Body jsou rozloženy rovnoměrně v intervalu <0,1> na přímce rovnoběžné s osou *x* nebo *y*. Další singulárním rozložení jsou data s body na oblouku kružnice. Ve třetí dimenzi patří mezi singulární případy množina vstupních bodů, jejichž souřadnice konvergují k povrchu na kouli. Ukázky singulárních případů jsou uvedeny na obrázku 7.2.



a) Body na křivce b) Body na kulové ploše

Obr. 7.2: *Příklad singulárních případů. Body na kulové ploše jsou zobrazeny v projekci do roviny yz.*

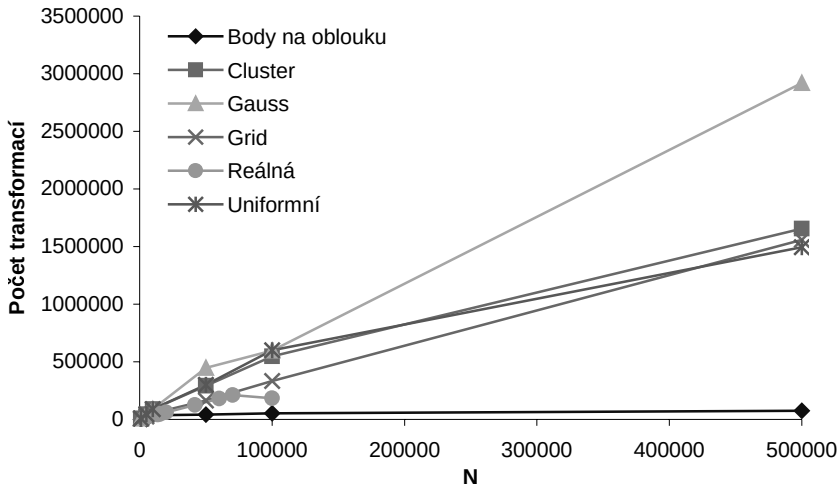
Kromě dat těchto uvedených rozložení mezi testovaná data patřila rovněž reálná data, ať již se jednalo o modely terénů (viz A.5) [Gar] či body populárních 3D povrchových modelů bunny, bell apod. [Stan]

7.1.1 Vliv rozložení

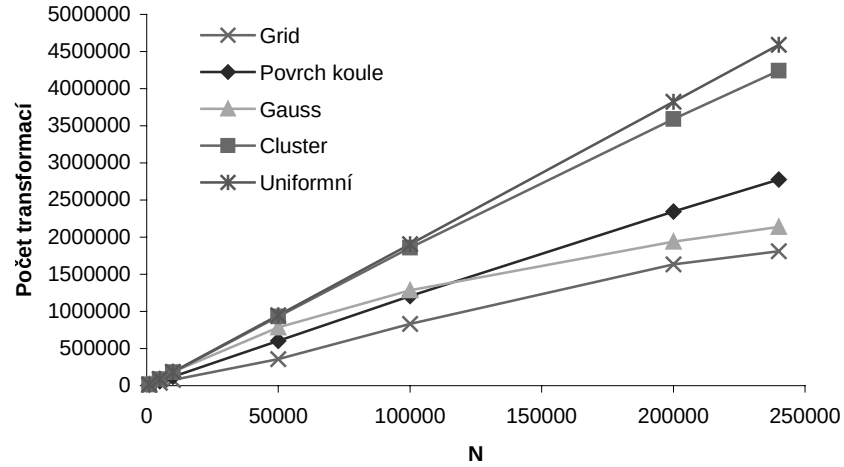
Je zřejmé, že typ vstupních dat má vliv jak na vyváženost vytvářené *DAG* struktury a tudíž i na velikost času potřebného pro vyhledání simplexu, tak i na počet nutných lokálních transformací a tudíž i na velikost času potřebného pro legalizaci. Zatímco pro trojrozměrný případ je výsledný čas určen výhradně počtem uskutečněných lokálních transformací, pro dvourozměrný případ se na výsledném čase projeví především, jak rozložení ovlivnilo velikost a vyváženost struktury (viz analýzy sériových algoritmů).

Počet transformací roste přibližně lineárně se zvyšujícím se počtem vstupních bodů *N*. Rozložení ovlivňuje linearitu i rychlost růstu (viz Graf 7.1). Na počtu uskutečněných lokálních transformací závisí počet vygenerovaných uzlů v *DAG*, tj. počet transformací nepřímo zpětně ovlivňuje (spolu s vyvážeností struktury) velikost času pro vyhledávání simplexu.

a) 2D případ



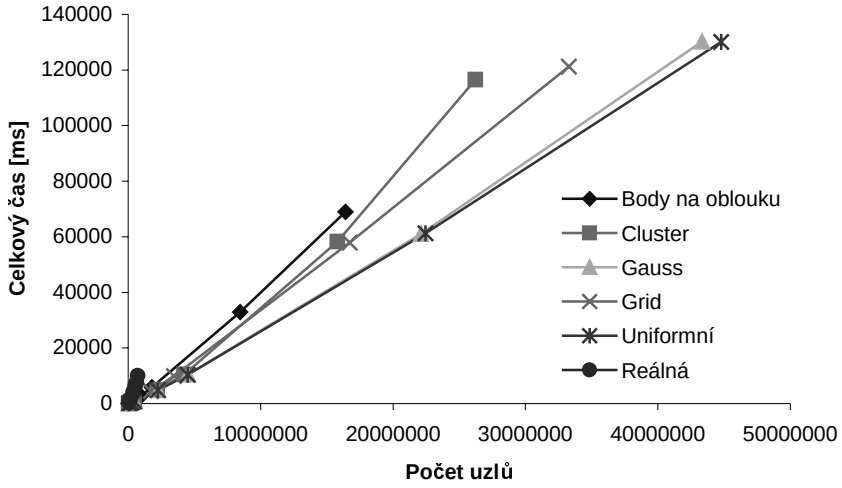
b) 3D případ



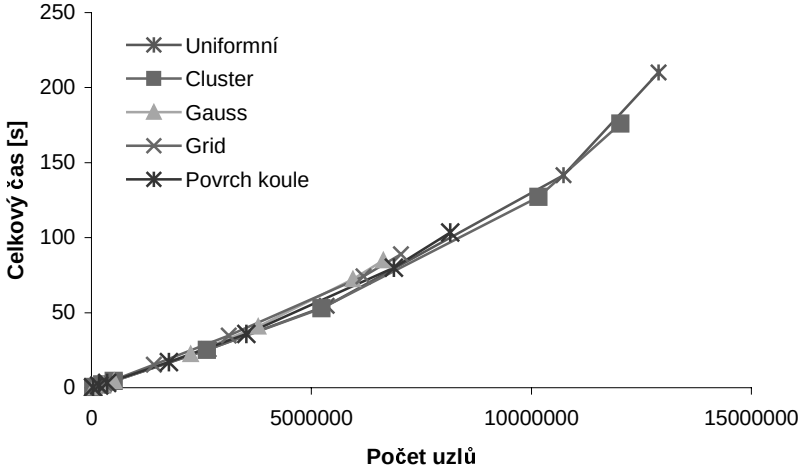
Graf 7.1: *Počet uskutečněných lokálních transformací v závislosti na počtu vstupních bodů N pro různé typy dat.*

Přesto se ukazuje, že pokud vyjádříme závislost dosaženého času sériové 3D verze na počtu vygenerovaných uzlů *DAG*), pak průběh této funkce pro všechny datová rozložení je stejný. Můžeme tedy závislost popsat jedinou funkcí nezávisle na typu vstupních dat (viz Graf 7.2b). Tuto závislost však nelze vypozorovat v případě sériové 2D verze (viz Graf 7.2a).

a) 2D případ



b) 3D případ



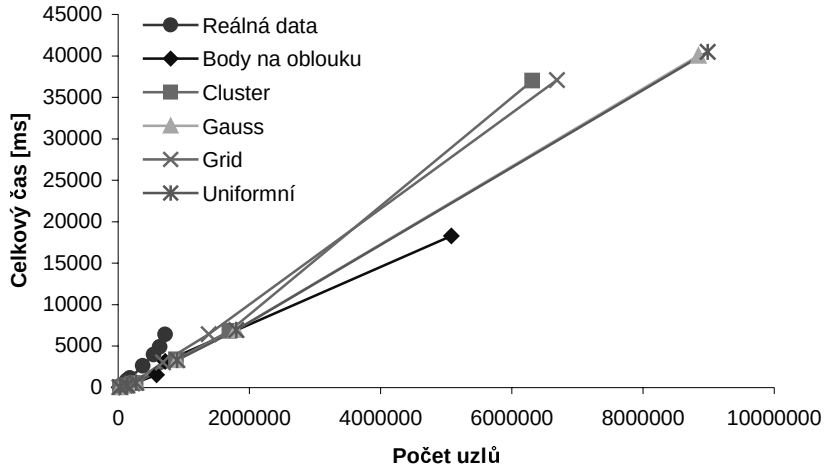
Graf 7.2: *Závislost celkového času na velikosti DAG struktury, tj. na počtu vygenerovaných uzlů v dvourozměrném (měřeno na 2x Intel Pentium Celeron 533Mhz, 512 MB RAM) a trojrozměrném (Dell Precision 410, 700MHz, 1GB RAM) případě sériové verze pro různá datová rozložení.*

Pro paralelní algoritmy obdržíme závislosti, které se příliš neodlišují od sériových verzí. Ve dvourozměrném případě (viz Graf 7.3a) se, změnil průběh funkce pro body na oblouku. Je to zapříčiněno tím, že vyhledávací fázi lze efektivněji paralelizovat než fázi legalizace, a právě tento typ dat (jak bylo ukázáno v grafu 7.1a) produkuje zanedbatelné množství lokálních transformací. Výsledkem je nižší celkový čas.

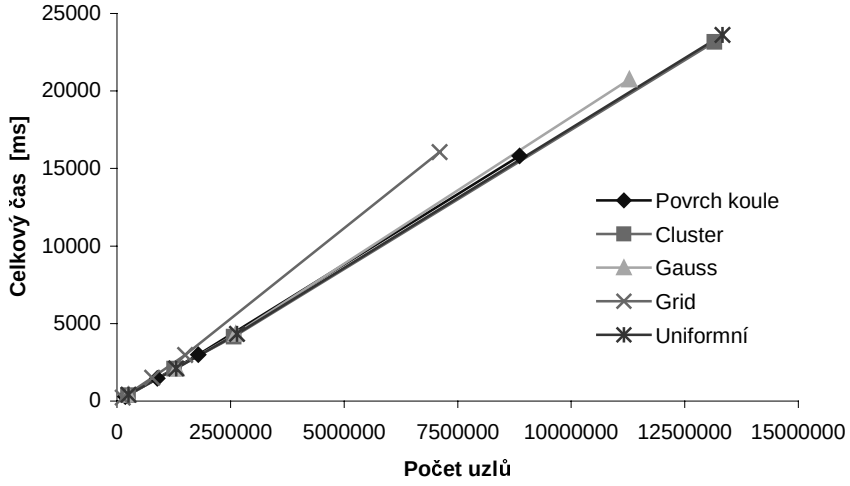
V grafu 7.3b si povšimněme náhlého odštěpení „gridových“ dat od společného průběhu. Vysvětlení je prosté. V kapitole popisující inkrementální metodu konstrukce *3D DT* jsme

uvedli čtyři možné lokální transformace: transformace dvou tetrahedronů na tři (*S23*), tří na dva (*S32*), dvou na dva (*S22*) a čtyř na čtyři (*S44*). Poslední dvě transformace mohou nastat, pokud čtyři body dvou sousedních tetrahedronů leží v jedné rovině. Tento případ je však kromě „gridového“ rozložení, kde transformace *S22* a *S44* představují asi 20%, vzácný. Z hlediska implementovaného uzamykajícího mechanismu je zamčení simplexů právě pro tuto transformaci několikanásobně časově náročnější než uzamčení simplexů pro *S23* či *S32*. Také zatímco pro uniformní data je případ vkládání nového bodu na hranu zanedbatelný, pro body uspořádané do pravidelné mřížky tvoří tento počet podstatné procento. Poznamenejme však, že takto pravidelná data se obvykle při konstrukci Delaunayovy triangulace nepoužívají.

a) 2D případ



b) 3D případ



Graf 7.3: *Závislost celkového času na velikosti DAG struktury, tj. na počtu vygenerovaných uzlů v případě optimistické verze s detekcí deadlocku pro různá rozložení. Měřeno na 2x Intel Celeron 533MHz.*

Fakt, že v trojrozměrném případě (s výjimkou „gridových“ dat) výsledný čas závisí přímo na počtu vygenerovaných uzlů, má hlavní význam v tom, že známe-li nebo dokážeme-li odhadnout u netestovaného typu vstupních dat počet uzlů, které v případě testu budou vygene-

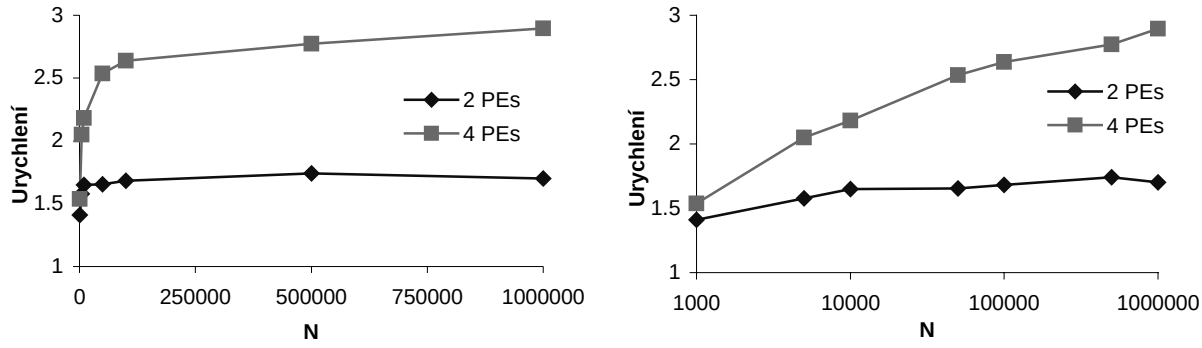
rovány, potom výsledek tohoto testu bude přibližně stejný jako známý výsledek testu jiného rozložení se stejným počtem vygenerovaných uzlů v *DAG* struktuře. Například víme, že při konstrukci *3D DT* množiny bodů s uniformním rozložení je počet vygenerovaných uzlů na jeden vkládaný bod průměrně 1.5krát vyšší než při použití množiny bodů rozložených na povrchu koule. Pro množinu 100 tisíc bodů na povrchu koule naměřený čas je 34.8 sekundy a vygenerovalo se zhruba 3.5 miliónu uzlů. Při uniformní rozložení s týmž počtem bodů by mělo vzniknout asi 5.25 miliónu uzlů, a dosaženo tedy času asi 52.2 sekundy. Provedeme-li měření, získáme výsledek: 5.35 miliónu uzlů a čas 54.7 sekundy.

Vzhledem k omezené možnosti otestovat aplikaci (na Západočeské univerzitě není k dispozici víceprocesorový počítač s Microsoft Windows NT 3.51 kompatibilním operačním systémem) jsme využili této charakteristiky k eliminaci množství testovacích datových množin. Jako dostatečně reprezentativní vzorek umělých dat jsme zvolili uniformní data. Protože ve dvourozměrném případě nelze provést obdobné zobecnění, uvedeme také výsledky z testování ostatních datových rozložení.

Poznamenejme, že pro reálná data (a to v obou případech, tj. jak v 2D tak i v 3D) roste celkový čas v závislosti na počtu vygenerovaných uzlů mnohem rychleji (viz Graf 7.2a a 7.3a pro 2D případ, ve 3D případě je charakteristika podobná charakteristice „gridových“ dat z paralelního testu, a to dokonce i v sériové verzi – ponecháno bez důkazu).

7.2 Pesimistický přístup

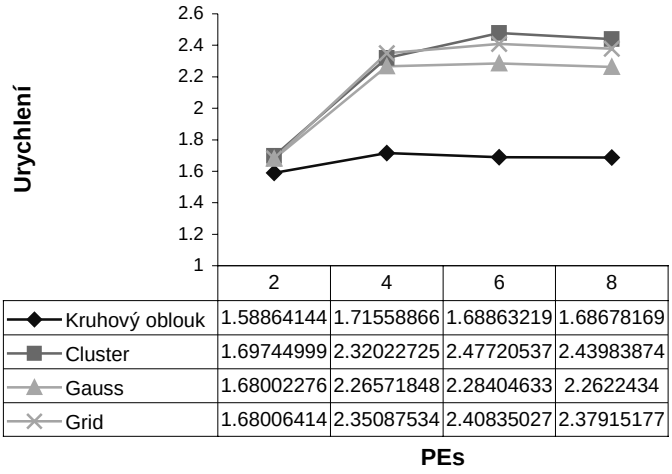
Pesimistický přístup je nejjednodušší, ale vzhledem ke kritické sekci (vlákno ve dvourozměrné variantě stráví 20% sériového času v chráněném kódu a ve trojrozměrné variantě dokonce okolo 65%) nelze očekávat zázračné urychlení pro vyšší počet procesorů. Přesto tuto metodu lze doporučit všem vlastníkům dvouprocesorových strojů, protože je rozumným kompromisem mezi jednoduchostí a urychlením. Dosažené urychlení této metody pro uniformní data uvádí graf 7.4.



a) Standardní měřítko b) Logaritmické měřítko na ose *x*

Graf 7.4: *Urychlení pesimistické metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny uniformních dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 7150.*

Graf 7.5 srovnává dosažené urychlení pro jednotlivé typy dat. Uniformní rozložení není v tomto grafu zahrnuto, průběh je blízký charakteristice normálního rozložení. Srovnání „gridových“ a uniformních dat lze nalézt v [Koh01]. Až na singulární případ bodů na kruhovém oblouku je výsledné urychlení téměř nezávislé na typu dat.



	2	4	6	8
Kruhový oblouk	1.58864144	1.71558866	1.68863219	1.68678169
Cluster	1.69744999	2.32022725	2.47720537	2.43983874
Gauss	1.68002276	2.26571848	2.28404633	2.2622434
Grid	1.68006414	2.35087534	2.40835027	2.37915177

Graf 7.5: *Urychlení pesimistické metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny 500000 bodů. Měření urychlení v závislosti na typu dat a procesorů provedeno na UniSys ES5000.*

Z obou grafů je zřejmé, že křivky urychlení nejprve prudce nelineárně rostou, pak stoupají téměř lineárně až k bodu, ve kterém použitý počet procesorů „nestíhá“ vyhledávat a urychlení zvolna klesá. Z tohoto důvodu je vhodné použít na zpracovávanou množinu dat pouze takový počet procesorů, který přinese optimální urychlení. Proto uživatel osmiprocesorového počítače nikdy nevyužije zcela jeho výkon, jelikož vhodná velikost datových množin již nebude zpracovatelná díky běžné 2GB paměťové bariéře 32-bitových aplikací.

Tabulka 7.1 shrnuje efektivitu pesimistické metody pro singulární případ bodů na kruhovém oblouku a pro ostatní datová rozložení (viz Graf 7.5), a srovnává ji s efektivitu vypočtenou na základě Amdahlova zákona o urychlení *S* paralelního kódu, který tvoří sekvenční část algoritmu *s* a paralelizovatelný kód *p*, přičemž s + p = 1 a *n* je počet procesorů:

$$S = \frac{s+p}{s+p/n} = \frac{1}{s+p/n} \tag{7.1}$$

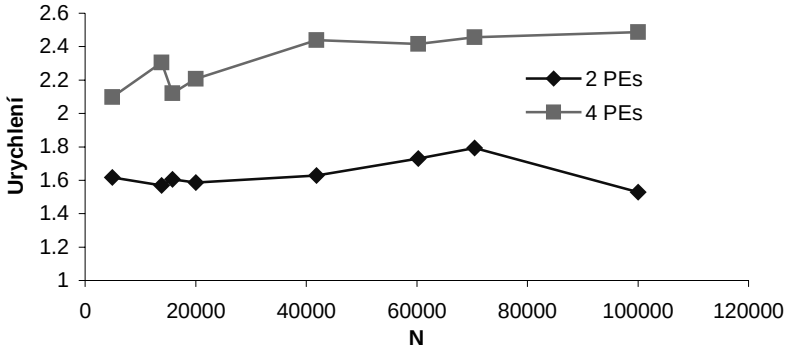
Z uvedené analýzy pro konstrukci 2D *DT* vyplývá, že při pesimistickém způsobu lze paralelizovat pouze asi 75% kódu. Proměnná *s* proto se rovná 0.25 a *p* = 0.75.

PEs	Singulární případ bodů na oblouku				Ostatní datová rozložení			
	Naměřené		Amdahl		Naměřené		Amdahl	
	Urychlení	Efektivita	Urychlení	Efektivita	Urychlení	Efektivita	Urychlení	Efektivita
2	1.57	78.5%	1.60	80.0%	1.67	83.5%	1.60	80.0%
4	1.53	38.3%	2.29	57.1%	2.24	56.0%	2.29	57.1%
6	1.51	25.2%	2.67	44.4%	2.31	38.5%	2.67	44.4%
8	1.50	18.8%	2.91	36.3%	2.29	28.6%	2.91	36.3%

Tabulka 7.1: *Průměrné urychlení a efektivita pesimistické metody pro konstrukci 2D DT s využitím EQ1 nad množinami s počtem bodů z intervalu <1000, 1mil> v porovnání s teoretickým urychlením a efektivitou spočtenou na základě Amdahlova zákona 7.1.*

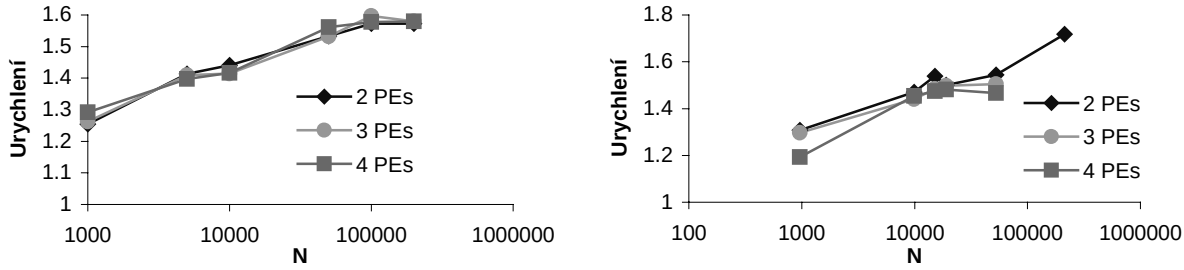
Protože umělá testovaná data nejsou dostatečně průkazná, byl algoritmus ověřen rovněž na reálných modelech terénů s počtem bodů 4897, 13829, 15820, 20014, 41853, 60244, 70433 a 100001 (Crater Lake – viz příloha C). Dosažené urychlení z grafu 7.6 je v souladu s očekáváním. Průměrné urychlení pro dva procesory je 1.63 (efektivita 82%), pro čtyři procesory je 2.32 (efektivita 58%), tedy žádné překvapivé zvraty.

Pesimistický způsob pro konstrukci Delaunayovy triangulace ve 2D lze doporučit uživatelům dvouprocesorových či čtyřprocesorových počítačů. V případě dvouprocesorového stroje urychlení vůči sériové verzi dosahuje 1.41 – 1.74, tj. efektivita se pohybuje od 70% po 87%. V případě čtyř procesorů urychlení dosahuje 1.54 – 2.90 (efektivita 38% – 73%). Nasazení vyššího počtu procesorů již nepřináší významné urychlení; efektivita je velmi nízká.



Graf 7.6: *Urychlení pesimistické metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny reálných dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 7150.*

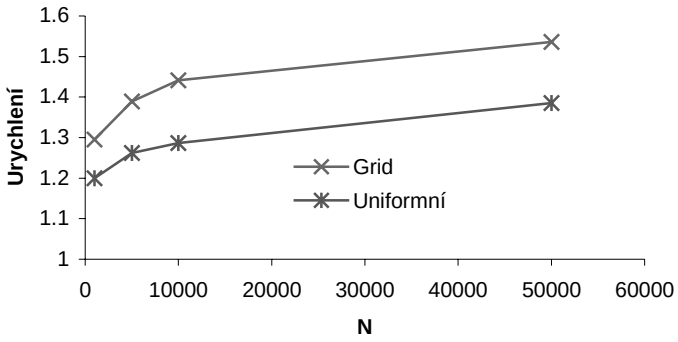
Zatímco velikost paralelizovatelného kódu ve dvourozměrné verzi činí asi 75%, v trojrozměrném případě dosahuje pouhých 30%. Graf 7.7 zachycuje urychlení pesimistické metody vůči sériové verzi. Je vidět, že nemá smysl uvažovat o metodě pro více než dva procesory. Naopak urychlení 1.25 – 1.57 v případě uniformních dat pro dva procesory (efektivita 63% až 79%) a urychlení 1.31 – 1.72 (66% – 86%) v případě reálných dat je nad naše očekávání. Z reálných dat byly testovány 3D modely cos (N = 961), montrose (9918), casual_man (15145), babysumo (19074), whale (52635), CTMayo(98869) a bell (213373).



a) Uniformní data b) Reálná data

Graf 7.7: *Urychlení pesimistické metody pro konstrukci 3D DT s použitím EQ1 distribuce vstupní množiny uniformních a reálných dat. Měření urychlení v závislosti na počtu bodů (měřítko na ose x logaritmické) a procesorů provedeno na Dell PowerEdge 7150.*

Urychlení poskytované metodou je v podstatě (obdobně jako ve 2D) nezávislé na typu vstupních dat a to i v případě singulárního případu bodů na povrchu koule. Výjimku tvoří „gridová“ data. Důvod byl již objasněn v předchozí podkapitole. Srovnání urychlení uniformního rozložení s „gridovým“ uvádí graf 7.8.



Graf 7.8: *Srovnání urychlení pesimistické metody pro konstrukci 3D DT s použitím EQ1 distribuce vstupní množiny uniformních a regulárních (grid) dat. Měření urychlení v závislosti na počtu bodů provedeno na 2x Intel Celeron 533MHz.*

Vzhledem k dosaženým výsledkům lze pesimistický způsob pro konstrukci Delaunayovy triangulace ve 3D doporučit jen uživatelům dvouprocesorových počítačů.

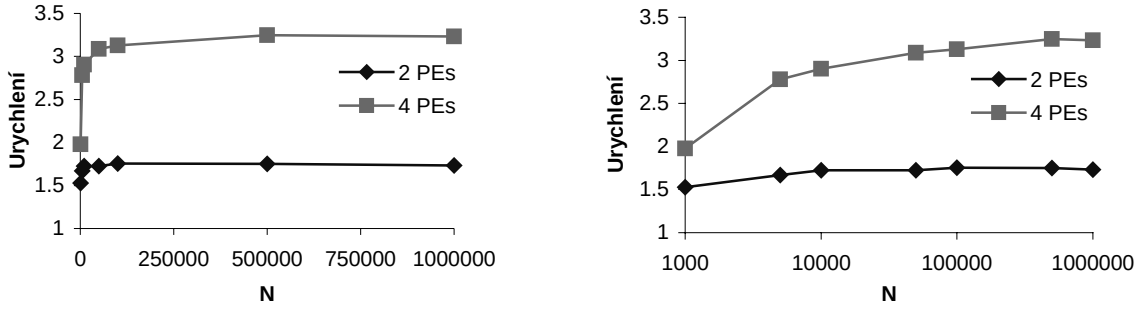
7.3 Optimistický přístup

V kapitole o paralelizaci jsme navrhli dvě optimistické metody: s detekcí deadlocku a s prevencí deadlocku. Zatímco optimistická metoda s detekcí deadlocku vyžaduje kratičký kód v kritické sekci, při optimistické metodě s prevencí se uskuteční mnohem více „rollback“ operací. Z hlediska praxe jsou tedy obě metody srovnatelné, ve 3D se ukázala nepatrně lepší metoda s detekcí deadlocku, ve 2D je tomu naopak. Srovnání obou metod jak pro uniformní data (poměr prevence/detekce je však stejný i pro jiná rozložení) jak ve 2D, tak ve 3D uvádí tabulka 7.2. Protože rozdíly jsou zanedbatelné a starší verze prokazovala horší výsledky metody s prevencí (vývojový důvod), nebudeme nadále optimistickou metodu s prevencí uvažovat a budeme-li mluvit o optimistické metodě, budeme mít na mysli vždy optimistickou metodu s detekcí deadlocku. Je nutné však poznamenat, že optimistická metoda s prevencí je nepatrně jednodušší a pro 2D nepatrně rychlejší.

2D				3D			
N	**Detekce**	**Prevence**	**Prev/Det**	***N***	**Detekce**	**Prevence**	**Prev/Det**
10000	585.651	608.839	1.040	1000	392.569	411.453	1.048
25000	1620.748	1598.565	0.986	5000	2136.299	2135.328	1.000
50000	3484.324	3465.941	0.995	10000	4241.344	4208.48	0.992
100000	7313.441	7301.063	0.998	50000	23749.7	23867.2	1.005
150000	11574.696	11529.285	0.996				1.011
200000	15976.734	15900.272	0.995				
			1.002				

Tabulka 7.2: *Čas potřebný na vložení všech bodů optimistickou metodou s detekcí v porovnání s časem spotřebovaným optimistickou metodou s prevencí. Měřena uniformní data na 2x Intel Celeron 533 MHz.*

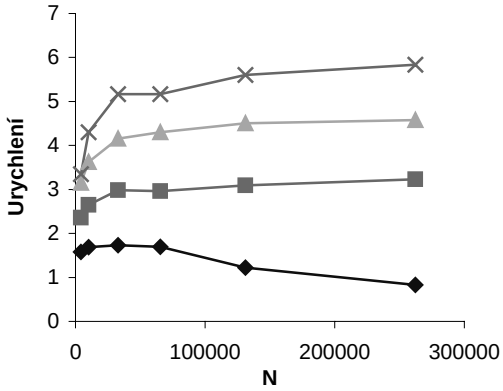
Procento paralelizovatelného kódu oproti 75% v 2D verzi pesimistické metody vzrostlo až na přibližně 92%, podle Amdahlova zákona tedy mezní urychlení pro dva procesory je 1.85 (efektivita 93%), pro čtyři procesory je 3.23 (efektivita 81%) a pro osm 5.13 (efektivita 64%). Praktické výsledky jsou uvedeny v grafu 7.9.



a) Standardní měřítko b) Logaritmické měřítko na ose *x*

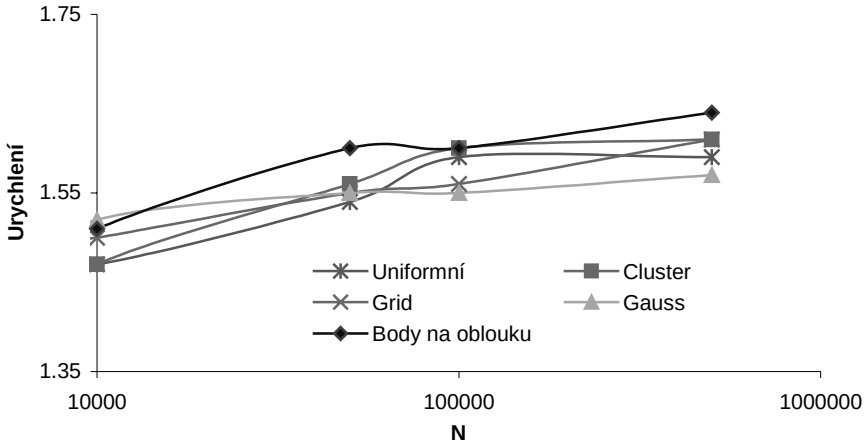
Graf 7.9: *Urychlení optimistické metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny uniformních dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 7150.*

Jak je patrné z tohoto grafu a dále pak z grafu 7.10 (hodnoty v něm uvedené však pocházejí z testování starší verze, navíc urychlení není vypočteno z celkových časů, ale jen z časů potřebných na vložení všech bodů), urychlení roste se zvyšujícím se počtem procesorů a se zvyšujícím se počtem bodů vstupní množiny. Výsledky prezentované v grafu 7.9 vykazují průměrné urychlení 1.75 pro dva procesory (tj. efektivita 88%) a průměrné urychlení 2.85 pro čtyři procesory (efektivita 71%). Pro přiblížení očekávaného urychlení metody na víceprocesorovém počítači můžeme využít hodnot grafu 7.10. Protože aktuální verze má menší zamykací režii a nižší podíl sekvenční části, urychlení vypočtené z celkových časů bude přibližně stejné jako urychlení starší verze. Optimistická metoda by měla dosáhnout průměrného urychlení 5.1 pro 8 procesorů (efektivita 63%). Tato hodnota je v souladu s očekávaným urychlením vypočteným z Amdahlova zákona.



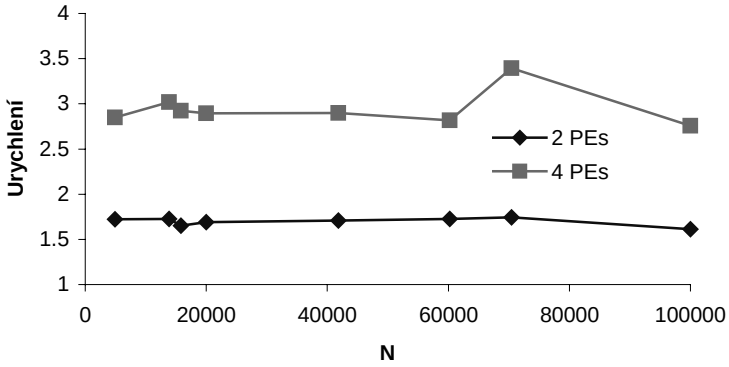
Graf 7.10: *Urychlení optimistické metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny regulárních (grid) dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 8450.*

Srovnání urychlení optimistické metody pro různá datová rozložení dokumentuje graf 7.11. Je patrné, že dosažená urychlení se příliš neliší.



Graf 7.11: *Urychlení optimistické metody pro konstrukci 2D DT s použitím EQ1 distribuce pro různé typy vstupních dat. Měření urychlení v závislosti na počtu bodů (logaritmické měřítko na ose x) provedeno na 2x Intel Celeron 533 MHz .*

Urychlení dosažené při testování reálných je prezentováno grafem 7.12. Průměrné urychlení pro dva procesory je opět 1.7 a pro čtyři procesory dokonce 2.96 (efektivita 74%).

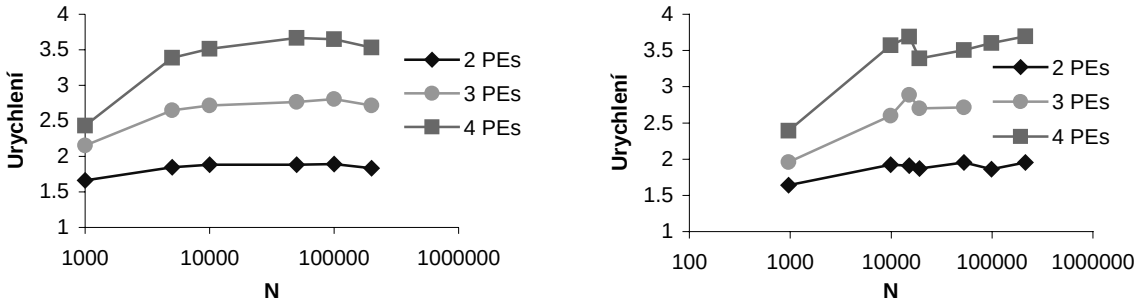


Graf 7.12: *Urychlení optimistické metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny reálných dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 7150.*

Optimistický způsob pro konstrukci Delaunayovy triangulace ve 2D lze doporučit všem uživatelům multiprocesorů s několika málo procesory. V případě dvouprocesorového stroje urychlení vůči sériové verzi dosahuje 1.53 – 1.75, tj. efektivita se pohybuje od 76% do 88%. V případě čtyř procesorů urychlení dosahuje 1.98 – 3.25 (efektivita 49% – 81%).

V trojrozměrné verzi podíl paralelizovatelného kódu nabývá hodnoty 97%, lze tedy očekávat vyšší urychlení než v případě 2D verze. Graf 7.13 zachycuje urychlení optimistické metody vůči sériové verzi. V případě uniformních dat se urychlení pohybuje v rozmezí 1.66 –

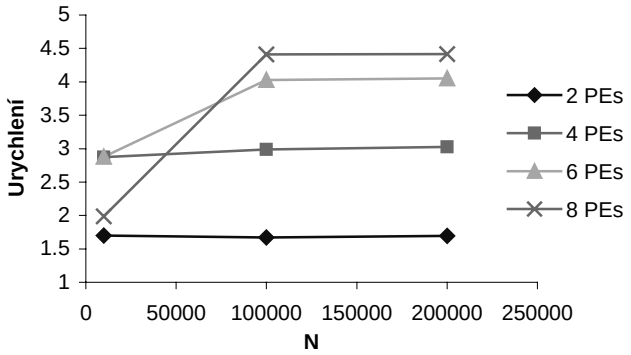
1.89 pro dva procesory (efektivita 83% až 95%) a pro reálná data je urychlení 1.64 – 1.96 (82% – 98%). Obdobné výsledky obdržíme při testování jiných datových rozložení. Pro čtyři procesory se dostáváme na 2.4 – 3.67 (60% – 92%) a v případě reálných dat na 2.39 – 3.69.



a) Uniformní data b) Reálná data

Graf 7.13: *Urychlení optimistické metody pro konstrukci 3D DT s použitím EQ1 distribuce vstupní množiny uniformních a reálných dat. Měření urychlení v závislosti na počtu bodů (měřítko na ose x logaritmické) a procesorů provedeno na Dell PowerEdge 7150.*

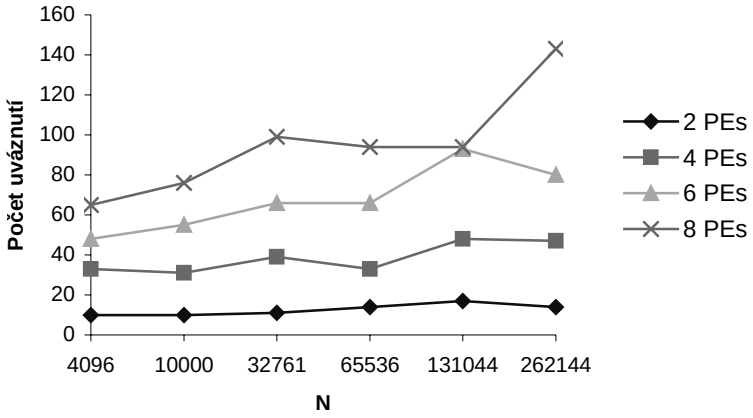
O chování metody při nasazení více procesorů vypovídá graf 7.14, který však opět poskytuje výsledky testu předchozí verze algoritmu. Zdá se, že maximálním rozumným počtem je 6 procesorů.



Graf 7.14: *Urychlení optimistické metody pro konstrukci 3D DT s použitím EQ1 distribuce vstupní množiny uniformních dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na UniSys ES5000.*

7.4 Zlodějská metoda

Graf 7.15 znázorňuje počet detekovaných uváznutí při běhu optimistické metody s detekcí deadlocku v 2D. Počet konfliktů roste velmi pomalu ve srovnání se zvětšující se vstupní množinou bodů. Pro regulární data s počtem bodů N = 262144 a osm procesorů deadlock se musí vyřešit ve 143 případech, zatímco 262001 bodů vloženo bez jediného problému. To představuje zbytečnost transakcí v 99.95%. Z tohoto důvodu byly vyvinuty metody, které eliminují transakce – např. zlodějská metoda.



Graf 7.15: *Počet detekovaných uváznutí v 2D v závislosti na počtu procesorů a počtu vstupních bodů. Měřena „gridová“ data na Dell PowerEdge 8450.*

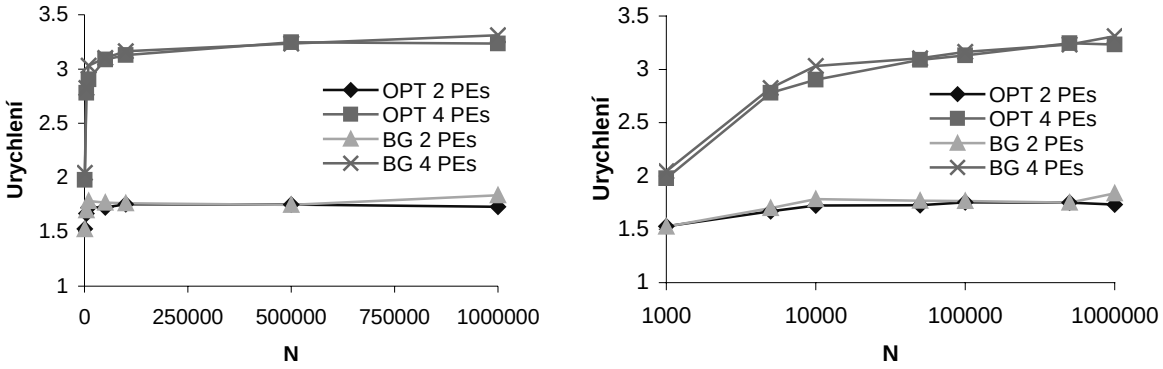
Zlodějská metoda, tak jak byla popsána v kapitole Paralelizace, navíc částečně eliminuje uzamykání. Testováním však bylo zjištěno, že uzamyká-li i zlodějská metoda všechny simplexy, ke kterým přistupuje, urychlení vzroste (viz Tabulka 7.3).

N	Stálé zamykání		Občasné zamykání	
	EQ1	EQ3	EQ1	EQ3
1000	1.488	1.503	1.483	1.505
10000	1.611	1.631	1.588	1.626
25000	1.641	1.643	1.606	1.629
50000	1.711	1.676	1.690	1.676

Tabulka 7.3: *Srovnání průměrného urychlení zamykací a nezamykací verze zlodějské metody pro konstrukci 3D DT s použitím EQ1a EQ3 distribuce vstupní množiny. Měřena regulární data, uniformní data a data s body na povrchu koule na 2x Intel Celeron 533 MHz.*

Při objasňování této záhady se ukázalo, že počet lokálních transformací, při nichž algoritmus se nacházel v neuzamykacím módu, ku počtu transformací provedených v uzamykacím módu nedosahuje ani celého procenta. Je patrné, že díky tomuto žalostnému poměru nelze očekávat žádné patrné urychlení. Přechod mezi jednotlivými módy však vyžaduje přídavnou synchronizaci. Kromě toho některé optimalizační úpravy použitelné při zamykání pro standardní optimistickou metodu, nelze aplikovat, pokud se neuzamyká stále. Další prezentované výsledky se budou vztahovat pouze na zlodějskou metodu se stálým zamykáním.

Výsledky testování algoritmu pro konstrukci *2D DT* jsou uvedeny v grafu 7.16 a grafu 7.17 (hodnoty v něm uvedené však pocházejí z testování starší verze, navíc urychlení není vypočteno z celkových časů, ale jen z časů potřebných na vložení všech bodů).

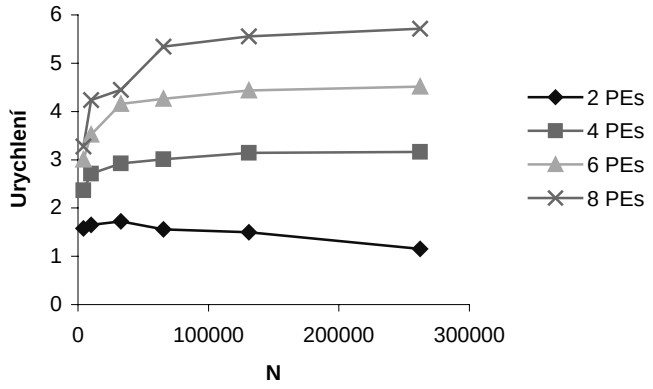


a) Standardní měřítko b) Logaritmické měřítko na ose *x*

Graf 7.16: *Srovnání urychlení zlodějské (BG) a optimistické (OPT) metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny uniformních dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 7150.*

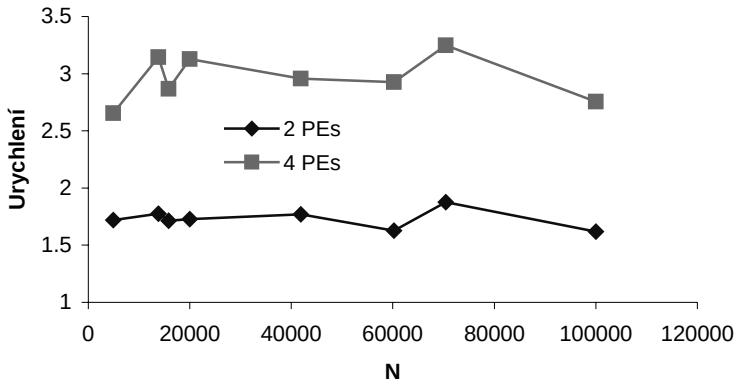
Urychlení se překvapivě příliš neliší od urychlení poskytnutého standardní optimistickou metodou, která navíc zaručuje bezchybnost sestrojené sítě. U zlodějské metody se jedná o průměrné urychlení 1.72 pro dva procesory (tj. efektivita 87%) a průměrné urychlení 2.90 pro čtyři procesory (efektivita 73%). Pro osm procesorů by zlodějská metoda měla dosáhnout průměrného urychlení 5.7 pro 8 procesorů (efektivita 63%).

Pravda, transakční mechanismus představoval asi dvě instrukce procesoru navíc pro každou modifikaci datové struktury, ale v průměru se jedná jen asi o 1000 instrukcí na jeden vkládaný bod. Na druhou stranu, ve zlodějské metodě je nutná verifikace aktuálně zpracovávaných uzlů, pokud činnost vlákna byla zlodějem narušena, a navíc uzamykací rutina je nepatrně složitější (ale je nutné si uvědomit, že tato rutina je volána ve 3D asi 27x, ve dvourozměrném případě je to o něco méně, pro každý vkládaný bod). To jsou také důvody, proč zlodějská metoda poskytuje pouze 1% urychlení vůči optimistické metodě.



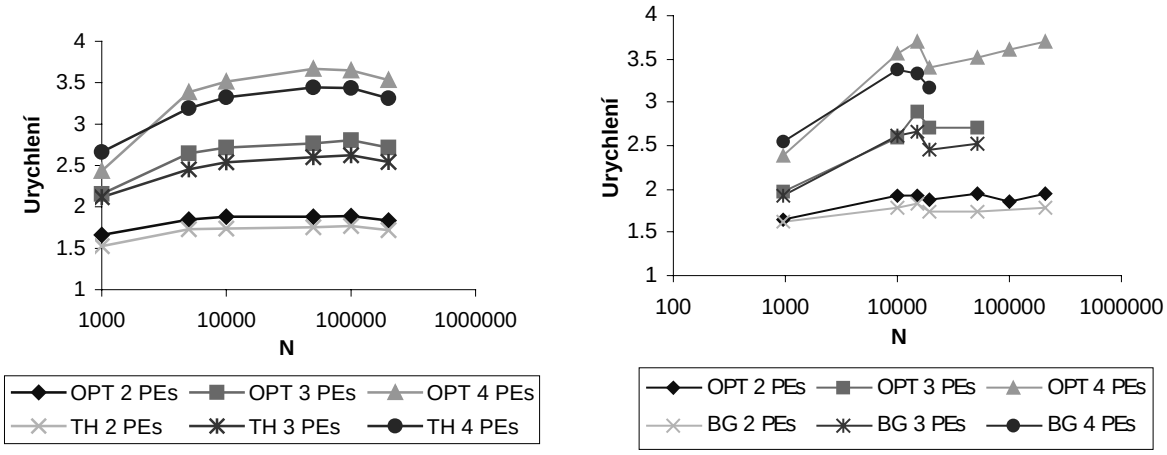
Graf 7.17: *Urychlení zlodějské metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny regulárních (grid) dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 8450.*

Přesuneme-li naši pozornost na reálná data (viz Graf 7.18), zjistíme, že průměrné urychlení je 1.73 pro dva procesory a pro čtyři 2.96.



Graf 7.18: *Urychlení zlodějské metody pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny reálných dat. Měření urychlení v závislosti na počtu bodů a procesorů provedeno na Dell PowerEdge 7150.*

Překvapením pro nás jsou výsledky testování 3D verze algoritmu (viz Graf 7.19). Zlodějská metoda nabývá v případě uniformních dat pro dva procesory urychlení 1.53 – 1.77 (tj. efektivita 77% až 89%); průměrné urychlení je 1.71 (efektivita 86%), zatímco optimistická metoda dosáhla průměrného urychlení 1.83. V případě reálných dat průměrné urychlení dosahuje hodnoty 1.75 (efektivita 88%), urychlení optimistické metody činí 1.89. Pro čtyři procesory se dostáváme na průměrné urychlení 3.23 (81%) v případě uniformních dat (optimistická metoda má průměrné urychlení 3.36). V případě reálných dat obdržíme výsledek 3.1 (efektivita 78%), zatímco optimistická metoda si drží hodnotu 3.48.



a) Uniformní data b) Reálná data

Graf 7.19: *Srovnání urychlení zlodějské (BG) a optimistické (OPT) metody pro konstrukci 3D DT s použitím EQ1 distribuce vstupní množiny uniformních a reálných dat. Měření urychlení v závislosti na počtu bodů (měřítko na ose x logaritmické) a procesorů provedeno na Dell PowerEdge 7150.*

7.5 Metoda vícenásobného zamykání a okamžitého odemykání

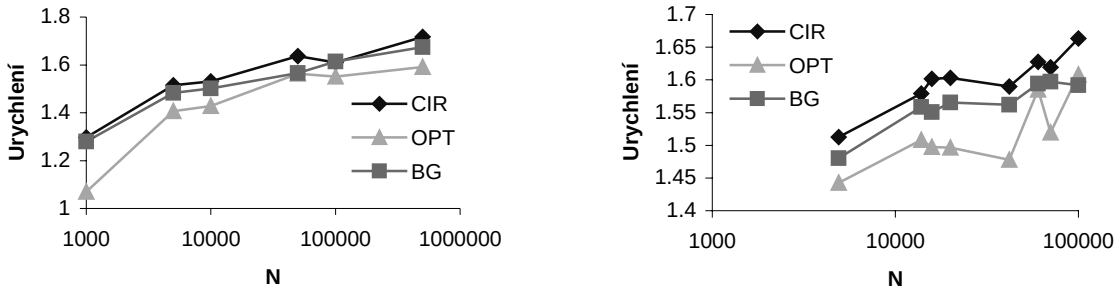
Od této metody se očekávalo, že přinese nejvyšší urychlení, protože eliminuje množství případů, kdy jedno vlákno musí čekat na druhé. Bohužel se tak nestalo. Ve 3D bylo dosaženo urychlení srovnatelného s pesimistickou metodou (leckdy dokonce horšího). Analýzou algoritmu se dospělo k následujícímu zdůvodnění. V průměru připadá na každý vkládaný bod přibližně 27 volání zamykací rutiny, během nichž je nutné získat přístup k celkem k 226 simplexům. 62.8% simplexů (označme je *AL*) z nich již bylo uzamčeno předchozími volání rutiny, tj. 37.2% se musí nově uzamknout. Většina z těchto nově uzamykaných simplexů ale není uzamčena konkurenčním vláknem (označme je *F*), simplex je blokován jiným vláknem pouze v 0.1% případů (označme tyto simplexy *W*). Tento výsledek byl dosažen nejen pro uniformní data (viz Tabulka 7.4), ale také pro jiná datová rozložení. Okamžitým odemykáním se tedy „nic“ neušetří, navíc přibude přídavná režie. Z tohoto důvodu nebyla metoda ve 2D vůbec implementována.

N	*MAIN*	*ALIN*	*FIN*	*WIN*
1000	25.80	134.62	79.95	0.202
10000	27.27	143.26	84.60	0.0321
25000	27.47	144.48	85.22	0.015216
50000	27.62	145.36	85.68	0.008452

Tabulka 7.4: *Analýza optimistické verze pro konstrukci 3D DT s použitím EQ1 distribuce. MA je počet uskutečněných volání zamykací rutiny, AL počet již uzamčených simplexů, F počet dosud volných a W počet simplexů vyhrazených pro konkurenční vlákno. Měřena uniformní data na 2x Intel Celeron 533 MHz.*

7.6 Metoda kružnice opsané

Jedná se o natolik novou metodu, že dosud nemohla být ověřena na jiných multiprocesorech než 2x Intel Celeron 533 MHz. Jak ukazují výsledky našich experimentů v grafu 7.20, metoda kružnice opsané dosáhla relativně významného urychlení oproti optimistické anebo zlodějské metodě. Obdobné srovnání bylo uskutečněno i pro ostatní rozložení, tato metoda je, s výjimkou singulárního případu bodů na oblouku (pro ně je nejvhodnější zlodějská metoda), nejrychlejší. Tato rychlost může být však vyvážena zvýšeným počtem chyb ve výsledné síti.



a) Uniformní data b) Reálná data

Graf 7.20: *Srovnání urychlení zlodějské (BG) a optimistické (OPT) metody a metody kružnice opsané (CIR) pro konstrukci 2D DT s použitím EQ1 distribuce vstupní množiny uniformních a reálných dat. Měření urychlení v závislosti na počtu bodů (měřítko na ose x logaritmické) provedeno na 2xIntel Celeron 533MHz.*

Tabulka 7.5 zachycuje počet případů, kdy bylo nutno vzdát se pokračování legalizace, a detekovaných vad ve výsledné síti. I přes tento počet je počet defektů až na výjimky nízký a v souladu s počtem defektů v síti vytvořené sekvenčním algoritmem (v tomto případě je počet defektů způsoben vlivem numerických chyb).

N	Body na oblouku			Cluster			Grid			Uniformní		
	Leg	CErr	SErr	Leg	CErr	SErr	Leg	CErr	SErr	Leg	CErr	SErr
1000	21	0	2	25	2	0	13	0	0	33	0	0
5000	18	0	0	43	6	0	43	0	0	66	0	0
10000	545	14	8	58	8	0	54	20	0	93	0	0
50000	716	14	14	86	4	6	142	0	0	183	6	0
100000	742	12	16	112	22	22	241	0	0	229	0	0
500000	911	18	24	234	3064	94	818	0	0	446	0	0

Tabulka 7.5: *Počet případů (Leg), kdy legalizace nebyla dokončena, maximální detekovaný počet chyb v síti vytvořené metodou kružnice opsané (CErr) a sekvenční metodou (SErr). Měření opakováno devětkrát na 2x Intel Celeron 533 MHz.*

Vystává otázka, proč tato metoda, která se zdá být velmi obdobná zlodějské metodě, produkuje pro některá rozložení vysoký počet defektů, ačkoliv zlodějská metoda ne. Vysvětlení spočívá v porovnání pravděpodobností, že vlákno se bude muset vzdát legalizace. Předpokládejme vlákna T_0 a T_1, která právě zahájila operace spojené s vložením bodů p_0 a p_1. Dále uvažujme simplex S_0. S_0 by ve fázi legalizace byl změněn buď vláknem T_0 (za předpokladu, že běh vlákna T_1 byl pozastaven) anebo vláknem T_1 (za předpokladu, že běh vlákna T_0 byl pozastaven). T_1 bude muset vstoupit do čekací smyčky s pravděpodobností r_0 (tato pravděpodobnost je nízká – majoritním případem pro zlodějskou metodu je případ, kdy veškeré operace spojené s vložením bodu p_0 vlákno T_0 dokončí dříve, než vláknem T_1 bude vznesen požadavek přístupu k simplexu S_1). S pravděpodobností r_1 dojde k uváznutí a s pravděpodobností r_2 přerušené vlákno nebude moci dokončit legalizaci. Pokud dojde k uváznutí, pravděpodobnost, že vlákno T_1 je přerušeným vláknem, je ½. Pravděpodobnost, že vlákno ve zlodějské metodě, bude nuceno se vzdát pokračování legalizace, je proto rovna $r_0 \cdot r_1 \cdot r_2 \cdot ½$. Pro metodu kružnice opsané dostáváme zcela odlišný výsledek. Vlákno T_0 nemůže beztrestně použít simplex S_0, protože v kružnici opsané tomuto simplexu leží bod p_1, vlákno musí počkat. Tudíž $r_0 = 1$ a, jak již bylo zdůvodněno, $r_1 = 1$. Výsledek je $r_2 \cdot ½$. Experimenty ukazují, že poměr případů, kdy se legalizace řádně nedokončila v metodě kružnice opsané, a případů nedokončené legalizace ve zlodějské metodě nezřídka bývá několik set.

Vzhledem k současné nerobustnosti metody (čas od času extrémní nárůst defektů) ji nelze obecně doporučit. Nabízí se možnost využít metodu v pipe-line zpracování, pokud výsledná síť bude obsahovat více defektů než povolený limit, síť se spočítá znovu (raději optimistickou metodou). I přes některé takto ztracené výpočty by z dlouhodobého hlediska měl být celkový čas nejnižší ze všech řešení.

7.7 Rozdělení množiny vstupních bodů mezi jednotlivá vlákna

Simplexy ve výsledné triangulaci vytvořené jedním vláknem tvoří jednu či více souvislých oblastí (mnohoúhelníky nebo mnohostěny). Tyto oblasti nazveme fragmenty. Pokud byla vstupní množina ideálně distribuována mezi jednotlivá vlákna, výsledkem by měla být jediná souvislá oblast, tj. vlákna si během své činnosti příliš nekonkurovala. Čím méně fragmentů a čím méně vrcholů fragmenty mají (tzn. čím menší je zastoupení hraničních simplexů v trian-

gulaci), tím nižší finální čas očekáváme, protože konkurování vláken bude nižší. Je zřejmé, že počet fragmentů vstoupá s rostoucím počtem procesorů.

V tabulkách 7.6 a 7.7 je uvedeno, jak navržené způsoby rozdělení množiny vstupních bodů (připomeňme ve stručnosti, že *EQ3* distribuce rozděluje vstupní body dle jejich *x*-ové souřadnice, *EQ5* dle $x^2 + y^2$ a EQ7 dle $x^2 + y^2 + z^2$) ovlivňují činnost algoritmu. Rozdíly mezi výsledky dosažené způsobem *EQ1* a jiným komplikovanějším způsobem (*EQ3*, *EQ5* resp. *EQ7*) jsou velmi výrazné zejména pro dvourozměrný případ, kde pro *EQ1* je počet fragmentů mnohonásobně větší než u *EQ3* nebo *EQ5*, konkrétně v případě užití 8 vláken více než 90% trojúhelníků leží u hranice (poznamenejme, že na osmiprocesorovém počítači bude situace ještě dramatičtější)!

Vláken	*N*	Počet fragmentů			Počet hraničních simplexů v %		
		EQ1	EQ3	EQ5	EQ1	EQ3	EQ5
2	1000	50.7	1.2	1.0	60.6	4.6	5.3
	10000	505.7	1.5	1.3	61.1	1.3	1.4
	25000	1289.0	1.5	1.5	61.4	0.7	0.9
	50000	2527.0	1.7	1.8	61.1	0.5	0.6
	100000	5101.5	2.3	2.2	61.2	0.4	0.5
	150000	7709.2	2.3	3.0	61.2	0.3	0.4
	250000	12742.0	3.7	3.7	61.2	0.2	0.3
	350000	17785.8	4.3	4.2	61.2	0.2	0.3
	500000	25329.3	5.2	4.7	61.2	0.2	0.2
8	1000	57.0	1.3	1.1	89.8	27.2	27.3
	10000	593.8	2.0	1.8	90.7	8.1	8.6
	25000	1479.0	1.9	1.9	90.6	5.1	5.4
	50000	2984.1	2.1	2.8	90.9	3.6	3.9
	100000	5973.3	3.4	3.1	91.0	2.6	2.8
	150000	9025.9	4.3	3.8	91.4	2.1	2.3
	250000	15145.4	5.1	4.8	91.7	1.6	1.8
	350000	21230.5	5.0	6.7	91.7	1.4	1.5
	500000	30339.0	6.6	7.3	91.7	1.2	1.3

Tabulka 7.6: *Vliv způsobu distribuce dat mezi vlákna ve 2D na počet vzniklých fragmentů a délku hranice. Výsledky jsou poskytnuté optimistickou metodou s detekcí deadlocku. Měřena uniformní data na 2x Intel Celeron 533 MHz.*

Vláken	*N*	Počet fragmentů				Počet hraničních simplexů v %			
		EQ1	EQ3	EQ5	EQ7	EQ1	EQ3	EQ5	EQ7
2	100	12.5	2.0	2.0	3.0	34.6	24.6	32.0	28.7
	1000	48.5	6.0	5.0	4.5	58.8	10.6	13.1	14.9
	10000	416	15.5	13.0	20.5	61.0	4.8	5.7	6.4
	25000	1017.5	19.5	32.5	35.0	61.1	3.4	4.2	4.9
	50000	2128.5	36.5	36.0	62.5	60.7	2.5	3.2	3.7
8	100	39.7	13.3	16.7	16.0	80.9	74.8	73.0	74.7
	1000	452.3	34.7	58.0	30.7	89.3	56.1	55.4	53.3
	10000	4910.7	80.7	83.0	67.0	90.7	29.1	29.3	26.1
	25000	12211.7	143.3	142.3	119.0	91.1	21.0	21.7	19.2
	50000	24698.0	211.3	247.7	196.0	91.3	16.3	17.8	15.7

Tabulka 7.7: *Vliv způsobu distribuce dat mezi vlákna ve 3D na počet vzniklých fragmentů a délku hranice. Výsledky jsou poskytnuté optimistickou metodou s detekcí deadlocku. Měřena uniformní data na 2x Intel Celeron 533 MHz.*

Metoda EBQ1 nesplnila ani z části naše očekávání. Nejen, že časově je příliš náročná, ale navíc vzniká počet fragmentů dokonce větší než v případě EQ1 (viz Tabulka 7.8)!

N	Počet fragmentů	
	EQ1	EBQ1
125	44	85
1000	318	965
8000	3616	6916
19683	9593	18013
29791	14625	29055

Tabulka 7.8: *Vliv způsobu distribuce EBQ1 ve 3D na počet vzniklých fragmentů v porovnání s EQ1. Výsledky jsou poskytnuté optimistickou metodou s detekcí deadlocku. Měřena „gridová" data na 2x Intel Celeron 533 MHz při použití 8 vláken.*

Zdá se tedy, že sofistikovanější přístup by mohl přinést ještě vyšší urychlení, než bylo dosud publikováno. Metody EQ3, EQ5 i EQ7 však vyžadují „setřídění" vstupní množiny bodů, tj. přídavný čas. Tabulka 7.9 uvádí srovnání časové složitosti algoritmu qsort z ANSI C knihovny a použitého algoritmu pro nalezení přibližného mediánu. Uvedené časy jsou v sekundách a urychlení je vypočteno jako podíl času potřebného QuickSort algoritmem (qsort) ku času potřebného námi použitým algoritmem pro nalezení přibližného mediánu.

N	Medián	qsort	Urychlení	*N*	Medián	qsort	Urychlení
5000	0.006	0.013	2.21	150000	0.146	0.598	4.09
10000	0.010	0.026	2.67	250000	0.242	1.052	4.36
25000	0.027	0.075	2.84	500000	0.481	2.233	4.65
50000	0.050	0.172	3.47	1000000	0.974	5.365	5.51
100000	0.103	0.380	3.68	2500000	2.616	16.647	6.36

Tabulka 7.9: *Srovnání časové složitosti rozdělení vstupní množiny pomocí medián a sortu; Intel Celeron 533Mhz, 256MB RAM*

Podívejme se konečně nyní, jak použití těchto metod distribuce ovlivní prezentované výsledky. Tabulka 7.10 srovnává celkové časy obdržené jednotlivými metodami distribuce při testování uniformních dat ve 2D. Ačkoliv způsoby distribuce *EQ3* či *EQ5* by zcela jistě u zlodějské metody měly přinést vyšší urychlení než *EQ1* (viz analýza fragmentace), praxe vykazuje naopak asi 2% zpomalení pro dva procesory. Totéž zpomalení vykazuje optimistická metoda a pesimistická metoda při použití *EQ5*. Neočekávaně pesimistická metoda při použití *EQ3* dosahuje 1% urychlení vůči *EQ1*. Nicméně vzhledem k výsledkům pesimistické metody toto urychlení nestojí za další zkomplikování algoritmu. Pro čtyři procesory (test množiny uniformních dat s 200000 a 500000 body probíhal na Dell PowerEdge 7150) se projevuje vliv úzkých geometrických oblastí u *EQ3* a zpomalení je ještě výraznější, až 18%. Naopak *EQ5* již vykazuje urychlení 0.5% vůči *EQ1* pro optimistickou metodu a asi 4% pro zlodějskou metodu. Přesto nelze rozhodně tento způsob doporučit všem uživatelům, protože doporučujeme optimistickou metodu a u ní dosažené urychlení 0.5% není příliš vysoké v porovnání se zvýšenými implementačními nároky.

N	Pesimistická metoda		Optimistická metoda		Zlodějská metoda	
	EQ1/EQ3	EQ1/EQ5	EQ1/EQ3	EQ1/EQ5	EQ1/EQ3	EQ1/EQ5
1000	0.95	0.99	1.04	1.01	1.04	1.01
5000	1.09	1.04	1.01	1.00	0.96	0.95
10000	1.01	0.97	0.93	1.00	0.96	0.99
50000	1.05	0.98	0.98	0.94	1.03	0.97
100000	0.98	0.97	0.89	0.97	0.97	0.97
500000	1.01	0.98	0.93	0.97	0.96	0.96

Tabulka 7.10: *Urychlení různých způsobů distribuce vstupní množiny bodů v E^2 vůči standardnímu způsobu EQ1. Měřena uniformní data na 2x Intel Celeron 533 MHz.*

Přesuneme-li se k trojrozměrným verzím algoritmů (viz Tabulka 7.11), obdržíme zcela odlišné výsledky. Největší průměrné urychlení bylo dosaženo pro EQ7, tj. distribuce vstupní množiny podle vzdálenosti bodu v prostoru od počátku souřadnic, a to asi 2% pro pesimistickou metodu, 4% pro optimistickou a 6% pro zlodějskou metodu. Nepatrně horší způsob distribuce EQ5 dosáhl urychlení 1%, 5% a 6%. Pro čtyři procesory (test množiny uniformních dat s 10000 a 25000 probíhal na Dell PowerEdge 7150) však žádné urychlení nebylo pozorováno, spíše naopak. Například EQ5 v průměru dosáhlo 5% zpomalení vůči EQ1.

N	Pesimistická metoda			Optimistická metoda			Zlodějská metoda		
	EQ1/EQ3	EQ1/EQ5	EQ1/EQ7	EQ1/EQ3	EQ1/EQ5	EQ1/EQ7	EQ1/EQ3	EQ1/EQ5	EQ1/EQ7
1000	1.00	1.01	1.03	1.06	1.05	1.02	1.03	1.02	1.02
5000	1.01	1.02	1.02	0.98	1.00	1.00	0.99	1.00	1.00
10000	1.00	1.01	1.03	1.15	1.14	1.14	1.21	1.21	1.22
50000	1.00	1.00	1.01	1.00	1.01	1.01	1.00	1.01	1.01

Tabulka 7.11: *Urychlení různých způsobů distribuce vstupní množiny bodů v E^3 vůči standardnímu způsobu EQ1. Měřena uniformní data na 2x Intel Celeron 533 MHz.*

Zdůvodnění, proč v praxi se nesystematické rozdělení množiny vstupních bodů (tj. způsob EQ1) ukazuje relativně nejlepším, ačkoliv teoreticky se zdá být málo efektivní, není složité. Jak již bylo prezentováno v tabulce 7.4 (pro dvourozměrný případ očekáváme obdobné výsledky, ne-li horší), počet konfliktů je v případě *EQ1* nízký, a i kdyby se použitím jiného způsobu distribuce snížil na desetinu, je tato změna zcela zanedbatelná. Jedinou změnou, kterou složitější způsob distribuce přinese, je zcela jiné pořadí vkládání bodů do triangulace. A pořadí vkládání ovlivňuje vyváženost vytvářeného stromu, a tudíž i časové nároky při vyhledávání simplexu. Randomizace pořadí vkládání je však citlivou záležitostí, proto pokud budeme opakovat několikrát měření téže množiny, někdy bude vyšší urychlení pro složitější způsob, někdy pro jednoduchý *EQ1*. Z tohoto důvodu (i přes publikované 6% urychlení ve 3D) a z důvodu jednoduchosti doporučujeme *EQ1* způsob distribuce. Proto také byl převážně tento způsob užíván v experimentech.

7.8 Shrnutí dosažených výsledků

Navrhli jsme několik paralelních metod a způsobů, jak rozdělit vstupní body mezi procesory. Dosažené nejlepší, nejhorší i průměrné výsledky shrnuje tabulka 7.12. Dle výsledků našich testů doporučujeme optimistickou metodu s detekcí deadlocku a prosté rozdělení množiny vstupních bodů (distribuce *EQ1*). Tato metoda je použitelná jak pro dvourozměrný pří-

pad, tak i pro případ trojrozměrný. Uživatelům dvouprocesorových počítačů lze rovněž doporučit pesimistickou metodu. Tato metoda, která je opět univerzální, je jednodušší, nicméně výsledky jsou horší.

		Pesimistická metoda		Optimistická metoda		Zlodějská metoda	
		Uniformní	Reálná	Uniformní	Reálná	Uniformní	Reálná
2D							
2 PEs	**Min**	1.41 (71%)	1.53 (76%)	1.53 (76%)	1.61 (81%)	1.53 (76%)	1.62 (81%)
	Max	1.74 (87%)	1.79 (92%)	1.75 (88%)	1.74 (87%)	1.84 (92%)	1.88 (94%)
	Ø	1.62 (81%)	1.63 (82%)	1.69 (85%)	1.70 (85%)	1.72 (86%)	1.73 (86%)
4 PEs	**Min**	1.54 (38%)	2.10 (52%)	1.98 (49%)	2.76 (69%)	2.04 (51%)	2.66 (66%)
	Max	2.90 (72%)	2.49 (62%)	3.25 (81%)	3.40 (85%)	3.31 (83%)	3.25 (81%)
	Ø	2.29 (57%)	2.32 (58%)	2.85 (71%)	2.96 (74%)	2.90 (73%)	2.96 (74%)
3D							
2 PEs	**Min**	1.25 (63%)	1.31 (66%)	1.66 (83%)	1.64 (82%)	1.53 (76%)	1.62 (81%)
	Max	1.57 (79%)	1.72 (86%)	1.89 (95%)	1.96 (98%)	1.77 (88%)	1.82 (91%)
	Ø	1.46 (73%)	1.52 (76%)	1.83 (92%)	1.89 (95%)	1.71 (85%)	1.75 (88%)
3 PEs	**Min**	1.26 (42%)	1.30 (43%)	2.16 (72%)	1.96 (65%)	2.12 (71%)	1.93 (64%)
	Max	1.60 (53%)	1.50 (50%)	2.81 (94%)	2.89 (96%)	2.62 (87%)	2.65 (88%)
	Ø	1.47 (49%)	1.45 (48%)	2.63 (88%)	2.61 (87%)	2.48 (83%)	2.43 (81%)
4 PEs	**Min**	1.29 (32%)	1.19 (30%)	2.40 (60%)	2.39 (60%)	2.66 (67%)	2.55 (64%)
	Max	1.58 (40%)	1.48 (37%)	3.67 (92%)	3.69 (92%)	3.44 (86%)	3.37 (84%)
	Ø	1.47 (37%)	1.42 (36%)	3.36 (84%)	3.48 (87%)	3.23 (81%)	3.10 (78%)

Tabulka 7.12: *Souhrnné srovnání dosažených výsledků navržených metod. Uvedeno je minimální, maximální a průměrné urychlení, v závorce vyznačena efektivita. Měřena uniformní a reálná data na Dell Power Edge 7150.*

Vzhledem k tomu, že někteří autoři dokazují správnost svých algoritmů pouze teoreticky, jiní používají speciální architektury s desítkami procesorů, je jakékoliv srovnání našich výsledků s výsledky existujících paralelních algoritmů složité. Z tabulky 7.13, která se o to přesto pokouší, vyplývá, že navržený algoritmus, jehož významnou předností je jednoduchost, příliš nezaostává za podobnými komplikovanějšími paralelními algoritmy (v některých případech je dokonce předčí).

PEs	2D		3D			
	OPT	Har–Ble	OPT	DeWall	InCode	Teng
2	1.75	1.82	1.66	1.70	1.79	1.85[3]
4	3.25	3.33	3.67	2.46	3.11	3.43[3]
8	5.71[1]	5.88	4.42[2]	3.05	5.31	6.08[3]

[1] Očekávané urychlení
[2] Původní urychlení, očekáváno vyšší
[3] Odhad urychlení vypočítaný na základě prezentovaného urychlení pro 128, 256 Pes

Tabulka 7.13: *Porovnání nejlepších výsledků optimistické metody s detekcí deadlocku (OPT) a nejlepších výsledků některých jiných paralelních algoritmů: Hardwick – Blelochův algoritmus (Har–Ble), algoritmus DeWall, InCode a Tengův algoritmus. Kromě výsledků Har–Ble, která pochází ze stroje se sdílenou pamětí, se jedná o výsledky distribuovaného výpočtu. Srovnáváno urychlení pro uniformní data (s výjimkou Tengova algoritmu, kde je rozložení neznámé).*

Závěr

8 Zhodnocení práce

Doby, kdy pouze výpočetní matematická centra si pořizovala pro své výpočty paralelní počítače, jsou pryč. Ceny počítačů, zejména v poslední době, se dlouhodobě drží na nízké úrovni, přibývá stále uživatelů, kteří si pořizují dvouprocesorové počítače za účelem zvýšení výkonu při běžných činnostech. Kromě výkonu vyžaduje dnešní uživatel také jednoduchost. A právě návrh jednoduchého paralelního algoritmu pro uživatele těchto počítačů s nízkým stupněm paralelismu byl cílem této práce, a tento cíl byl splněn.

V této práci bylo navrženo několik jednoduchých paralelních algoritmů pro konstrukci Delaunayovy triangulace ve 2D a 3D metodou inkrementálního vkládání. Všechny navržené algoritmy byly implementovány za použití vývojových nástrojů Microsoft Visual C++ 6.0 a Microsoft Visual Studio.NET 7.0, a to jak v podobě samostatně spustitelného programu, tak jako modul do MVE. Ke své činnosti využívají DLL knihovnu se sériovou verzi algoritmu pro konstrukci *DT* implementovanou Doc.Dr.Ing. Ivanou Kolingerovou v Borland Delphi 5.0. Je vyžadována x86 kompatibilní paralelní architektura se sdílenou pamětí a operačním systémem Microsoft Windows NT 3.51+, Microsoft Windows 2000 či Microsoft Windows XP.

Implementované řešení bylo otestováno jak na uměle vygenerovaných datech různých distribucí (uniformní, regulární, gauss, cluster, body na kruhovém oblouku či povrchu koule), tak na reálných datech (modely terénů, klasické povrchové 3D modely). Dosažené urychlení je téměř nezávislé na typu dat, z navržených metod lze jednoznačně doporučit tzv. optimistickou metodu se detekcí deadlocku, která je relativně jednoduchá a použitelná v 2D i 3D.

V dvourozměrném případě bylo dosaženo průměrného urychlení 1.7 – 2.93 pro 2 – 4 PEs, očekáváno 5.7 pro 8 PEs. Toto urychlení je srovnatelné s výsledky existujícího paralelního Hardwick-Blelochova algoritmu (1.8 – 5.8), který je však značně složitý a navíc ho nelze přenést do 3D. V trojrozměrném případě bylo dosaženo průměrného urychlení 1.85 – 3.39 pro 2 – 4 PEs. Toto urychlení předčilo populární DeWall algoritmus, který se chlubí výsledky 1.7 – 3.35 pro 2 – 16 PEs. Srovnatelným soupeřem je InCoDe algoritmus (1.79 – 19.01 pro 2 – 64), který není příliš složitý a lze ho provozovat i na architekturách s distribuovanou pamětí. Detailní srovnání těchto několika algoritmů (bohužel autoři ostatních paralelních algoritmů své praktické výsledky neuveřejnili, někteří dokazují správnost algoritmu pouze teoreticky) naleznete v poslední podkapitole předchozí kapitoly. Urychlení implementované optimistické metody je pravděpodobně přijatelně škálovatelné až do osmi procesorů. Ověření (hlavně pro 3D verzi) se však dosud neuskutečnilo, protože možnost přístupu k osmiprocesorovému počítači je značně omezená a závisí na získání přístupu na osmiprocesorový počítač, který vlastní firma Dell Computer, Praha, ČR (asi jednou za půl roku).

Pro odstranění závislosti možnosti testování na interních podmínkách v komerčních firmách (tyto podmínky nemůžeme ovlivnit), lze očekávat, že další případné práce budou zaměřeny spíše na distribuované prostředí.

8.1 Seznam publikací

[Koh01] Kohout J.: Parallel Incremental Delaunay Triangulation, *Proceedings of the 5th Central European Seminar on Computer Graphics*, str. 85 – 94, 2001

[Kol00] Kolingerová I., Kohout J.: Pessimistic Threaded Delaunay Triangulation by Randomized Incremental Insertion, *Graphicon 2000*, str. 76 – 83, Russia, Moscow, 2000

Literatura

Seznam literatury použité pro tuto práci

[Ble96] Blelloch G.E., Miller G.L., Talmor D.: Developing a Practical projection-Based Parallel Delaunay algorithm, *Proceedings of the 12th Annual Symposium on Computational Geometry,* ACM, 1996

[Chr96] Chrisochoides *N*., Sukup F.: Task Parallel Implementation of the Bowyer-Watson Algorithm, *Proc. of the 5 th International Conference on Numerical Grid Generation in Computational Fluid Dynamic and Related Fields,* 1996

[Chr98] Chrisochoides *N*., Nave D., Hawblitzel C.: Data Migration Framework for the Load Balancing of Parallel Adaptive Unstructured Mesh Computations, *Proceedings of the 6th International Conference on Numerical Grid Generation in Computational Field Simulation,* University of Geenwich, Avery Hill Campus, London, UK, 1998

[Chr99] Chrisochoides *N*., Nave D.: Simultaneous Mesh Generation and Partitioning for Delaunay Meshes, *Proc. of the 8 th International Meshing Roundtable,* South Lake Tahoe, CA, USA, str. 55 – 66, říjen 1999

[Cig92] Cignoni P., Montani C., Perego R., Scopigno R.: Parallel 3D Delaunay Triangulation, Internal Report C92/20, listopad 1992

[Cig93] Cignoni P., Montani C., Perego R., Scopigno R.: Parallel 3D Delaunay Triangulation, *Computer Graphics Forum (Eurographics'93),* str. C129 – C142, 1993

[Har97] Hardwick J.C.: Implementation and Evaluation of an Efficient Parallel Delaunay Triangulation Algorithm, *9th Annual Symposium on Parallel Algorithm and Architectures*, str. 22-25, 1997

[Jež99] Ježek K., Matějovic P., Racek S.: Paralelní Architektury a Programy, Západočeská univerzita, Plzeň, 1999

[Joe91] Joe B.: Construction of three-dimensional Delaunay triangulations using local transformations, Computer Aided Geometric Design, Volume 8, str. 123 – 142, 1991

[Koh01] Kohout J.: Parallel Incremental Delaunay Triangulation, *Proceedings of the 5th Central European Seminar on Computer Graphics*, str. 85 – 94, 2001

[Kol99] Kolingerová I.: Rovinné triangulace, habilitační práce, Západočeská univerzita, Plzeň, 1999

[Kol00] Kolingerová I., Kohout J.: Pessimistic Threaded Delaunay Triangulation by Randomized Incremental Insertion, *Graphicon 2000*, str. 76 – 83, Russia, Moscow, 2000

[Lee97] Lee F.: Constructing the Constrained Delaunay Triangulation on the Intel Paragon, *Proceedings of the 13th Annual Symposium on Computational Geometry*, str. 464 – 467, ACM, 1997

[Mag98] Magillo P., Puppo E.: Algorithms for parallel terrain modelling and visualisation, *Parallel Processing Algorithms for GIS*, str. 351 – 386, 1998

[Mau00] Maur P.: Generování tetrahedronových sítí z rozptýlených dat, diplomová práce, Západočeská univerzita, Plzeň, 2000

[Ten93] Y.A. Teng , F. Sullivan, I. Beichl. and E. Puppo. A Data Parallel Algorithm for Three-dimensional Delaunay Triangulation and Its Implementation, *Super-Computing '93*, str.112-121, 1993.

[Žár98] Žára J., Beneš B., Felkel P.: Moderní počítačová grafika, Computer Press, 1998

Seznam doporučené literatury

[Ber97] de Berg M., van Kreveld M., Overmars M., Schwarzkopf O.: *Computational Geometry. Algorithms and Applications,* Springer–Verlag Berlin Heidelberg, 1997

[Dwy86] Dwyer. R.A.: A Simple Divide-and-conquer Algorithm for Constructing Delaunay Triangulation in *O(n log log n)* expected time. *Proc. of the 2nd Annual Symposium on Comp. Geom.*, pp. 276-284, ACM, 1986

[For87] Fortune S.: A sweepline algorithm for Voronoi diagrams. *Algorithmica*,2: str. 153–174, 1987

[Gui85] Guibas L.J., Stolfi J.: Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams, ACM Transactions on Graphics, Vol. 4, No. 2, 1985, str. 75-123

[Ost99] Ostromoukhov V., Hersch R.D.: Sochastic Clustered-Dot Dithering, *Color Imaging: Device-independent Color, Color Hardcopy, and Graphic Arts IV, SPIE Vol. 3648,* str. 496 – 505, 1999

[Pra00] Prasad L., Rao L.R.: A Geometric Transform for Shape Feature Extraction, Mathematical Imaging, 4117 Vision Geometry IX, *Proc. of the 45th SPIE Annual Meeting,* San Diego,CA, 2000

[Rup94] Ruppert J.: A Delaunay Refinement Algorithm Quality 2-Dimensional Mesh Generation, in *Journal of Algorithms*, 1994

[Su94] Sue P.: Efficient Parallel Algorithms for Closest Point Problems,

[Žal01] Žalik B., Kolingerová I., Podgorelec D.: An Incremental Construction Algorithm for Delaunay Triangulation Based on Two-level Uniform Subdivision, submitted to *International Journal of Geographical Information Science*, 2001

Odkazy

[Gar] Garland, M,: Sample data for terrain simplification, http://graphics.cs.uiuc.edu/~garland/research/quadrics.html

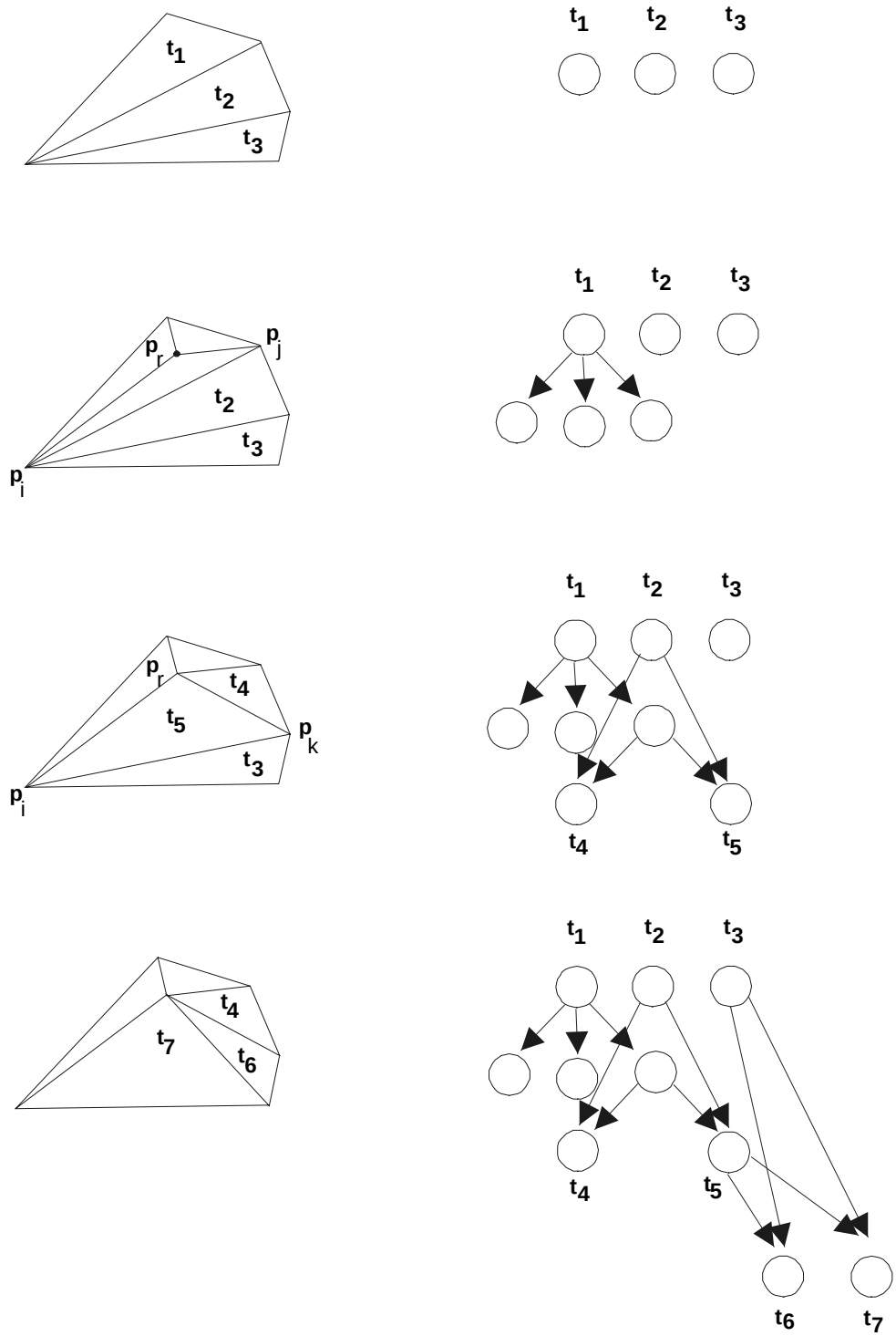
[Stan] Databáze 3D modelů Stanford http://www-graphics.stanford.edu/data/3Dscanrep/

[MVE] Modular Visualization Environment http://herakles.zcu.cz/research.php

Domácí stránky tohoto projektu se nachází na této URI:

http://herakles.zcu.cz/research/pdt/index.php

Příloha A



a) Změny v triangulaci b) odpovídající změny v *DAG* struktuře

Obr. A.1: *Propagace prohazování diagonál ve 2D DT a odpovídající změny v DAG struktuře.*

```
procedure Delaunay_triangulation
     In:  množina P = {p_i, i = 0,1,...,N-1} bodů v rovině
     Out: Delaunayova triangulace DT(P)

1. begin
2.  Sestroj velký počáteční trojúhelník
3.  Spočti náhodnou permutaci p_0, p_1, ..., p_n-1 z P;
4.  for r := 0 to n-1 do  begin   // vložení p_r do DT(P)
5.        Lokalizuj trojúhelník p_i p_j p_k ∈ DT(P) obsahující p_r;
6.         if p_r leží uvnitř p_i p_j p_k then begin
7.               Přidej hrany z p_r do bodů p_i p_j p_k a
                       rozděl  p_i p_j p_k  na tři trojúhelníky;
8.               Legalize_edge(p_r ,p_i p_j,DT(P));
9.               Legalize_edge(p_r ,p_j p_k,DT(P));
10.              Legalize_edge(p_r ,p_k p_i,DT(P));
11.         end else begin  // p_r  leží na hraně p_i p_j p_k , např. p_i p_k
12.               Přidej hrany z p_r do p_j a do p_l , třetího vrcholu
                      trojúhelníka sdílejícího hranu p_i p_k, a rozděl
                      dva trojúhelníky sdílející p_i p_k na čtyři;
13.               Legalize_edge(p_r ,p_i p_j,DT(P));
14.               Legalize_edge(p_r ,p_j p_k,DT(P));
15.               Legalize_edge(p_r ,p_k p_l,DT(P));
16.               Legalize_edge(p_r ,p_l p_i,DT(P));
17.         end;
18.  end;
19.  Odstraň všechny trojúhelníky obsahující vrcholy
           velkého počátečního trojúhelníka;
20. end


procedure Legalize_edge(p_r ,p_i p_j,T);
1. begin
2.  if p_i p_j is illegal then begin
      // p_i p_j p_k  je trojúhelník sdílející hranu p_i p_j s  p_r p_i p_j
3.    Zaměň p_i p_j za  p_r p_k
4.    Legalize_edge (p_r ,p_i p_k,T)
5.    Legalize_edge (p_r ,p_k p_j,T)
6.  end
7. end
```

Algoritmus A.2: *Sekvenční verze algoritmu konstrukce 2D Delaunayovy triangulace inkrementálním vkládáním randomizovaných vstupních bodů zapsaná v pseudo-Pascalu [Ber97].*

```
procedure Delaunay_3D_triangulation
In:  množina P = {p_i, i = 0,1,...,n-1} bodů v prostoru
Out: Delaunayova triangulace DT(P)

1. begin
2.  Sestroj velký počáteční tetrahedron
3.  Spočti náhodnou permutaci p_0, p_1, ..., p_n-1 z P;
4.  for r := 0 to n-1 do  begin   // vložení p_r do DT(P)
5.        Lokalizuj tetrahedron p_i p_j p_k p_l ∈ DT(P) obsahující p_r;
6.        if p_r leží uvnitř p_i p_j p_k p_l then begin
7.              Rozděl p_i p_j p_k p_l  na čtyři tetrahedrony;
8.              Legalize_face(p_r ,p_i p_j p_k,DT(P));
9.              Legalize_face(p_r ,p_i p_j p_l,DT(P));
10.              Legalize_face(p_r ,p_i p_k p_l,DT(P));
11.              Legalize_face(p_r ,p_j p_k p_l,DT(P));
12.        end else
13.        if p_r leží na stěně p_i p_j p_k p_l, např. p_i p_j p_k then begin
14.              Rozděl p_i p_j p_k p_l a sousední čtyřstěn p_i p_j p_k p_m
                       na celkem čtyři tetrahedrony;
15.              Legalize_face(p_r ,p_i p_j p_l,DT(P));
16.              Legalize_face(p_r ,p_i p_k p_l,DT(P));
17.              Legalize_face(p_r ,p_j p_k p_l,DT(P));
18.              Legalize_face(p_r ,p_i p_j p_m,DT(P));
19.              Legalize_face(p_r ,p_i p_k p_m,DT(P));
20.              Legalize_face(p_r ,p_j p_k p_m,DT(P));
21.        end else begin // p_r leží na hraně p_i p_j p_k p_l, např. p_i p_j
22.              Rozděl všechny tetrahedrony sdílející hranu p_i p_j na
                       dva a jejich stěny, které se musí ověřit,
                       vlož do seznamu M;
23.              while M není prázdný do begin
24.                    Legalize_face(p_r ,M.stěna,DT(P));
25.                    Odstraň první prvek seznamu M;
26.              end;
27.        end;
28.  end;
29.  Odstraň všechny tetrahedrony obsahující vrcholy
            velkého počátečního tetrahedronu;
30. end

procedure Legalize_face(p_r ,p_i p_j p_k,T);
1. begin
2.  if p_i p_j p_k is illegal and is swapable then begin
3.  //triangulace je neplatná, ale lze provést lokální změnu
4.    Podle příslušné 3D konfiguraci, proveď záměnu a nové stěny,
5.       které se musí ověřit, vlož do seznamu M;
6.    while M není prázdný do begin
7.          Legalize_face(p_r ,M.stěna,DT(P));
8.          Odstraň první prvek seznamu M;
9.    end;
10.  end;
11. end
```

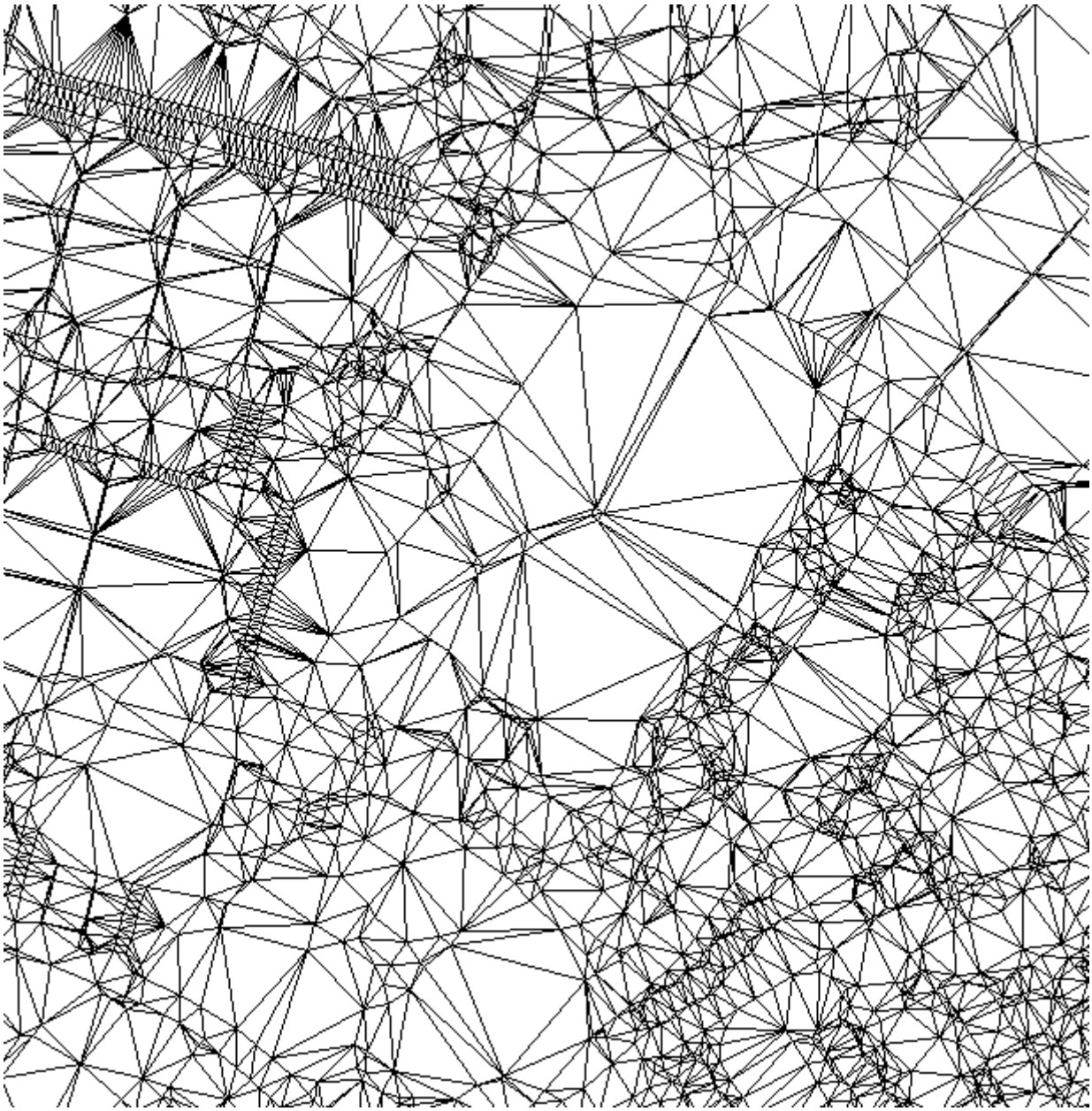
Algoritmus A.3: *Sekvenční verze algoritmu konstrukce 3D Delaunayovy triangulace inkrementálním vkládáním randomizovaných vstupních bodů zapsaná v pseudo-Pascalu.*

```
Function DeWall(P: množina_bodů, AFL: seznam_stěn) : Triangulace
  var f, fn : stěna; AFL^α, AFL^+, AFL^- : seznam_stěn;
      t : simplex; DT:  seznam_simplexů; α: dělící_rovina;
      P^+, P^- : množina_bodů;

1.  begin
2.   AFL^α, AFL^+, AFL^- := []; DT := [];   //v seznamech není žádná hrana
3.   α := SelectPlane();                    //dělící rovina se cyklicky volí
                                            //vždy rovnoběžně s x, y, z
4.   Pointset_Partition(P, α, P^+, P^-);   //a dělí vstupní množinu na dvě
5.   if AFL=[] then begin
6.      t := Make_First_Wall_Simplex(P, α)
        AFL := Faces(t); Insert(t, DT);
           //inkrementální konstrukcí se sestrojí první simplex
           //hrany sestrojeného simplexu vložíme do seznamu stěn,
           //vytvořený simplex zahrneme do výsledné triangulace
7.  end;
8.  //rozděl stěny v AFL do tří tabulek
9.  for each f in AFL do
10.   if Intersection(f, α) then Insert(f, AFL^α);
11.   else if f in NegHalfspace(α) then Insert(f, AFL^-);
12.   else if f in  PosHalfspace(α) then Insert(f, AFL^+);
13. while AFL^α <> []  //dokud není tabulka protínajících stěn prázdná
14.   f := Extract(AFL^α);     //vyber stěnu a k ní
15.   t := Make_Simplex(f,P)  //zkonstruuj další simplex
16.   if t <> NULL then begin //byl-li simplex úspěšně vytvořen
17.      Insert(t, DT);       //vlož simplex do výsledné triangulace
18.      for each fn in Faces(t)and fn <> f do
                        //a jeho další stěny zpracuj
                        //pokud se zpracovává stěna poprvé, vloží se do
                        //příslušného seznamu, jinak se z něj odebere
19.         if Intersection(fn, α) then Update(fn, AFL^α);
20.         else if fn in NegHalfspace(α) then Update(fn, AFL^-);
21.         else if fn in  PosHalfspace(α) then Update(fn, AFL^+);
22.   end;
23.  //aplikuj rekurzivně DeWall na obě množiny bodů
24.  if AFL^- <> [] then DT := DT ∪ DeWall(P^-,AFL^-);
25.  if AFL^+ <> [] then DT := DT ∪ DeWall(P^+,AFL^+);
26.  DeWall := DT;
27. end; //end DeWall

Procedure Update(f: stěna; L: seznam_stěn);
begin
  if f in L then Delete(f, L);
  else Insert(f, L);
end;
```

Algoritmus A.4*: DeWall algoritmus zapsaný v pseudo-Pascalu.*



Obr. A.5: *Ukázka části triangulace množiny bodů z kolekce reálných dat. Na obrázku model terénu.*

Příloha B

Programová realizace

Řešení vychází z existujícího modulu do MVE, který konstruuje *DT* sekvenčním způsobem. Tento modul je však implementován v Borland Delphi, zatímco zde popisovaný program je napsán v programovacím jazyce C++. Z tohoto důvodu bylo nutné navrhnout jednoduchý komunikační rozhraní nejen mezi dvěma moduly, ale také mezi C a Pascalem.

Pascalská knihovna DTLib se sekvenční verzí exportuje funkce a procedury potřebné paralelním programem. Tyto funkce a procedury musí dodržovat konvenci volání *stdcall*. Korespondující hlavičky těchto rutin v C++ jsou umístěny v souboru PDT.h.

K vytvoření komunikačního rozhraní, tj. zjištění adres jednotlivých vyexportovaných rutin, dochází v C++ programu vzápětí poté, co je knihovna zavedena do paměti. Stará se o to metoda *CPDT::PrepareInterface*.

Hlavní program na základě uživatelské volby vyplní inicializační strukturu a předá ukazatel na ni knihovně DTLib zavoláním funkce *Initialize_SDT*. Tato funkce, podle hodnot v předané struktuře, nastaví privátní vnitřní proměnné a načte nebo vygeneruje množinu vstupních bodů. Hlavní program poté spouští měření času a zahajuje výpočet. Zavolá funkci *Initialize_DTProc*, která provede sekvenční inicializaci výpočtu zahrnující randomizaci pořadí vkládání, nalezení počátečního simplexu a vytvoření kořene *DAG* struktury. Funkce vyplní tzv. běhovou strukturu (adresu na ni obdržela jako parametr) adresami *DAG* struktury, pole souřadnic a pole s pořadím vkládání. Po návratu z funkce program vytvoří příslušný počet pracovních vláken a počká na skončení jejich běhu. Volána je funkce *Finalize_DTProc*, která z *DAG* struktury extrahuje triangulaci a uvolní alokovanou paměť pro výpočet. Je ukončeno měření času a volána funkce *Finalize_SDT*, která ověří platnost vzniklé triangulace.

Implementační detaily

Je nutné, aby paměť alokovala Delphi DLL knihovna, proto byla vytvořena funkce *GetNewDAGItem*, která alokuje paměť pro nový uzel *DAG* struktury. Operace vzniku nového uzlu je velmi častá, a tak se může stát, že dvě vlákna ve stejnou chvíli potřebují alokovat nějakou paměť. V Borland Delphi 2.0 volání *new* není atomické, a proto vlákna mohou obdržet překrývající se bloky paměti. Aplikace dříve či později havaruje. Každé volání musí být tedy ochráněno kritickou sekcí. Jenže kritická sekce zpomaluje činnost paralelního programu, a jak ukazuje obrázek B.1, její využití pro optimistickou metodu je nešťastné. Z tohoto důvodu paralelní program alokuje vždy v kritické sekci více uzlů naráz a později je nechráněně přiděluje. Počet uzlů, které se mají alokovat, aby efektivita byla maximální, je alchymií. Tento počet byl stanoven na 512.

Transakční zásobník, do kterého se ukládají uskutečněné zápisy, je statickou strukturou a navíc se nekontroluje, zda je ještě v něm místo. Tento přístup sice vede k maximální možné efektivitě transakcí, nicméně pokud je zásobník přečerpán, aplikace havaruje. Testováním ve 2D bylo stanoveno, že pro libovolná vstupní data se nikdy neuskuteční více než 1000 transakcí pro jeden bod. Z tohoto důvodu stanovená velikost zásobníku 4096 snižuje pravděpodobnost krachu na nulu. Situace ve 3D je o mnoho horší, neboť pro nepříznivá data počet transakcí vyroste leckdy až na 200 tisíc. Tento extrémní nárůst je dán numerickou nestabilitou sériového algoritmu. Například pro skupinu tří tetrahedronů je vyhodnoceno, že se má provést swap23 a o chvíli později se legalizace opět vrátí k této skupině tří tetrahedronů, ale tentokrát se uskuteční swap32. Toto zacyklení, pokud ho přeruší změna triangulace vyvolaná vložením jiného bodu konkurenčním vláknem, způsobí nárůst transakcí. Pokud není zacyklení přerušeno, dojde k vyčerpání paměti a selhání výpočtu (v nejhorším případě k havárii programu).

Protože zásobník o 200K položek je zbytečně veliký v naprosté většině případů, je transakční zásobník řešen dynamicky. Počáteční velikost zásobníku je opět 4096 položek. Paralelní algoritmus v legalizační fázi pravidelně testuje, zda zásobník není již ze 60% plný. Pokud ano, zvětší ho na dvojnásobnou velikost.

Obdobně jako paralelní algoritmus používá transakční zásobník pro transakce, používá ještě další zásobník pro uzamčené simplexy. Opět ve 2D je řešen staticky (velikost 8192 položek) a v trojrozměrném případě dynamicky (počáteční velikost rovněž 8192 položek).

K označení „kdo simplex uzamkl“ slouží horní polovina 32-bitové proměnné *Flags* v uzlu *DAG* struktury, dolní polovina určuje typ uzlu (viz pesimistická metoda). Horní polovina jednoduše obsahuje číslo vlákna + 1, tj. hodnota 1 znamená, že uzel uzamklo vlákno s číslem 0; hodnota nula znamená volný uzel. Uzamčení se musí provést atomicky, ale bez kritické sekce. Nejprve je nutné přečíst hodnotu *Flags* požadovaného uzlu. Pokud je uzel obsazen, musí se počkat. Čeká se na stav, kdy se hodnota proměnné *Flags* změní (to je umožněno direktivou *volatile*). Dokud hodnota je stejná, vlákno se periodicky vzdává svého zbývajícího časového kvanta (*yield*) ve prospěch ostatních. Toto pseudo-aktivní čekání je zvoleno z důvodu, že čekání je vždy zanedbatelně krátké a navíc jakákoliv synchronizace spojená s neaktivním čekáním vede k vyšším časům paralelního algoritmu. V okamžiku dokončení čekání je uzel potenciálně volný, atomická funkce *InterlockedCompareExchange* (instrukce CMPXCHG) ověří zda tomu tak vskutku je, a pokud ano, změní hodnotu proměnné *Flags* tak, aby uzel byl vláknem uzamčen. Předběhl-li vlákno někdo jiný, volání funkce selže a vlákno se musí vrátit zpět do čekací smyčky a počkat na další příležitost. Je zřejmé, že k takto popsanému synchronizačnímu mechanismu je nutné přidat ještě detekci a řešení deadlocku (viz obr. B.2).

V případě metody vícenásobného zamykání musí být synchronizační mechanismus rafinovanější, neboť kromě vlastního zámku je nutné měnit i počet zámků a obě změny se musí provést v rámci jedné atomické operace. Implementované řešení (bez kritické sekce) není zcela korektní, asi ve 2% případů dojde k uváznutí aplikace. Vzhledem k praktickým výsledkům metody se však nejedná o tragédii.

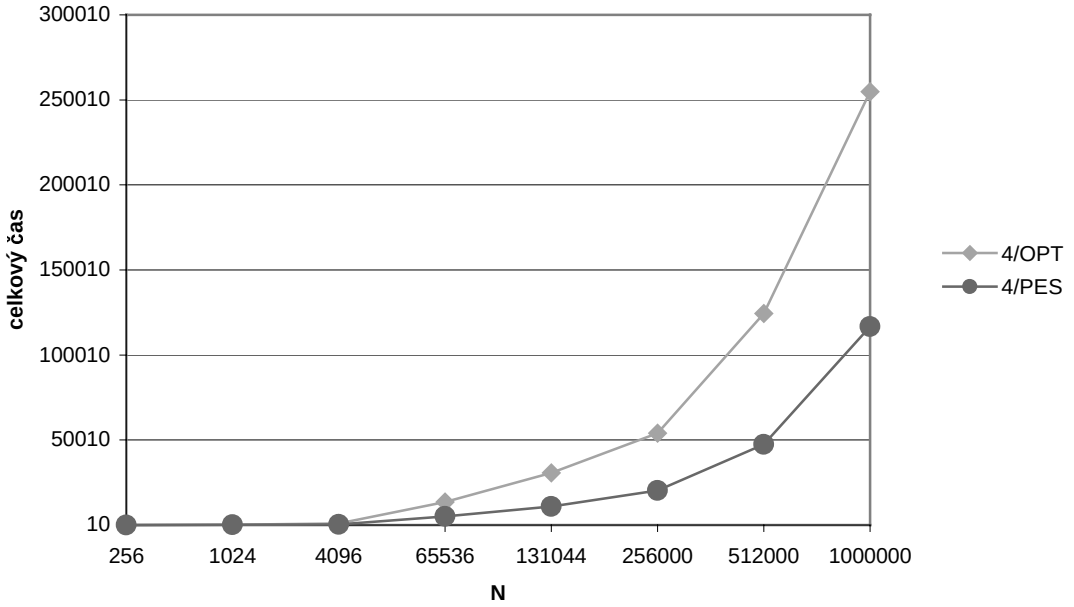
Programové moduly

Jádrem paralelního modulu je třída CPDT odvozená od MFC základní třídy CObject. Tato třída definovaná v PDT.h zapouzdřuje inicializaci DTLib knihovny (soubor PDT.cpp), sekvenční konstrukci Delaunayovy triangulace (soubor PDT.cpp) a vlastní paralelní konstrukci *DT* (rozdělena do několika souborů; pesimistická verze je v souboru Pess1.cpp, optimistická v Opt.cpp, zlodějská v Thief.cpp, další metody jsou uvedené v souborech Thief2.cpp, Circle.cpp, Sphere.cpp, Batch.cpp).

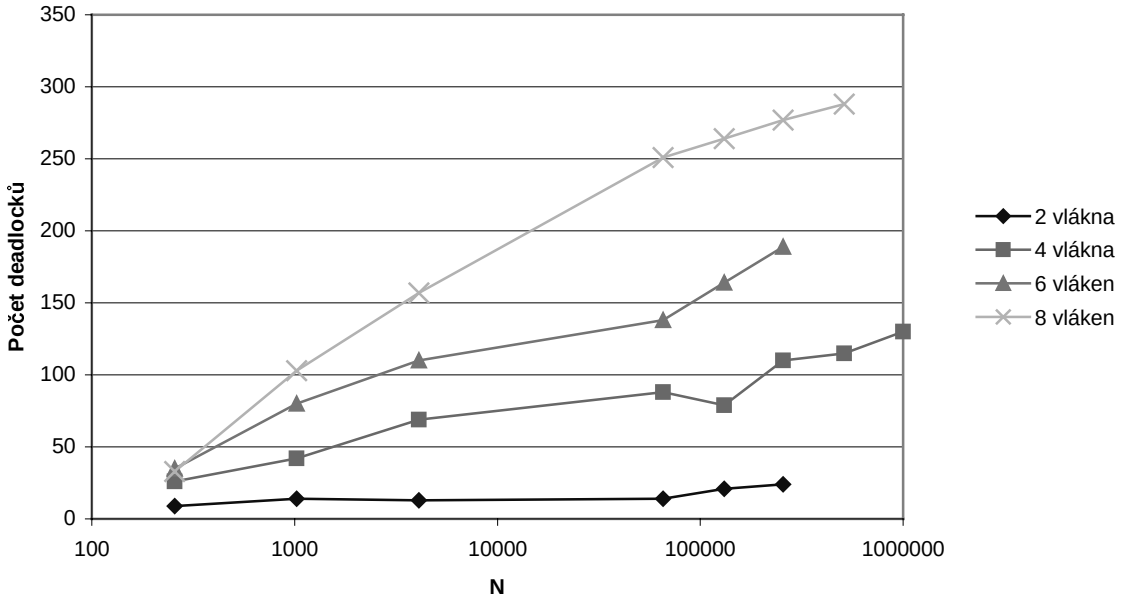
Další významnou třídou je CMainDlg odvozená od MFC třídy CDialog a zapouzdřující dialog pro nastavení výpočtu. Tato třída se nachází v souborech MainDlg.cpp a MainDlg.h.

Pro samostatnou aplikaci slouží ještě úvodní dialog CParallel_DTDlg (soubory Parallel_DTDlg.cpp a Parallel_DTDlg.h), ve kterém se aplikace konfiguruje. Z tohoto dialogu lze rovněž spustit výpočet pomocí skriptu, který lze napsat v dialogu CScriptDlg. Tato třída je obsažena v souborech ScriptDlg.cpp a ScriptDlg.h.

Pro verzi pro MVE místo těchto dvou tříd existuje soubor Parallel_DTLib.cpp, který obsahuje komunikační interface mezi modulem a systémem MVE.



Obr. B.1: *Srovnání optimistické metody s detekcí deadlocku (OPT) s pesimistickou metodou (PES). Zamykání řešeno přes kritickou sekci, alokace paměti postupná a rovněž přes kritickou sekci. Měřena uniformní data na Dell PowerEdge 8450 (použity 4 procesory).*



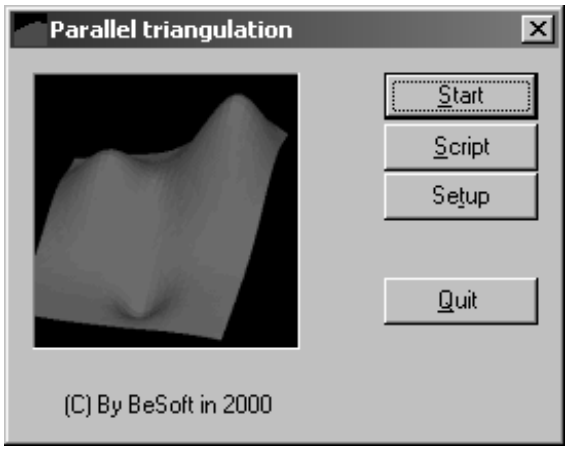
Obr. B.2: *Počet detekovaných uváznutí optimistickou metodou s detekcí deadlocku v závislosti na počtu procesorů a počtu bodů (logaritmické měřítko osy x) . Měřena uniformní data na Dell PowerEdge 8450.*

Uživatelská dokumentace (aplikace)

Úvodní dialog

Úvodní dialog (viz obr. B.3) se zobrazí po spuštění aplikace. Obsahuje tato čtyři tlačítka:

- Setup – specifikuje umístění DLL knihovny se sekvenční verzí. Pokud uživatel nemá tuto knihovnu umístěnou ve stejném adresáři jako aplikaci, je nezbytné vyvolat tuto funkci před zahájením prvního výpočtu. Při opětovném spuštění aplikace již lze provést výpočet okamžitě.
- Start – vyvolá dialog pro nastavení samostatného výpočtu.
- Script – vyvolá editor skriptu, načte do něj soubor *script.dts* a umožní uživateli změnit jeho obsah. Tento skript pak řídí tzv. dávkové zpracování a umožňuje zkonstruovat tisíce triangulací bez nutné interakce s uživatelem.
- Quit – řádné ukončení aplikace. Veškerá nastavení budou uložena na disk do souboru *Parallel_DT.ini* a načtena automaticky při opětovném spuštění aplikace.



Obr. B.3: *Úvodní dialog aplikace.*

Dialog pro samostatný výpočet

Tento dialog (viz obr. B.4) nastavuje parametry výpočtu a umožňuje uživateli výpočet zahájit. Obsahuje tři sekce:

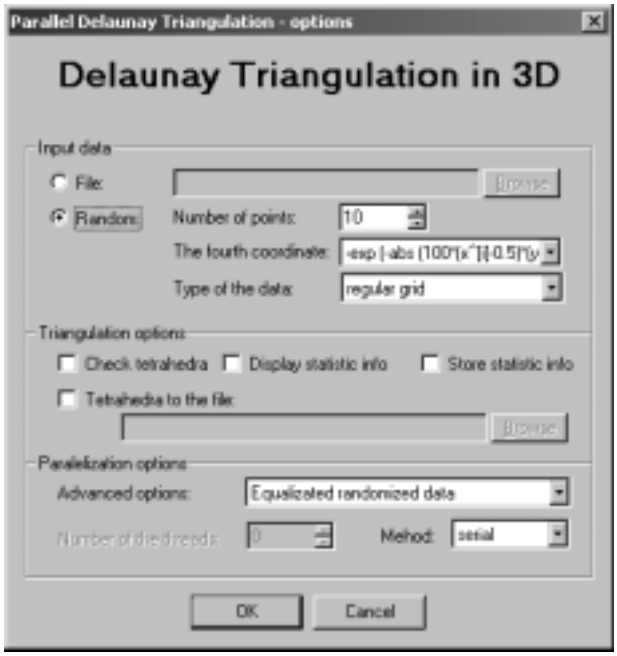
- Input data – v této sekci uživatel specifikuje, zda množina vstupních bodů bude načtena ze souboru nebo vygenerována interním generátorem DLL knihovny. V případě volby generovat náhodná data si uživatel volí typ dat mezi pravidelnou mřížkou, uniformními daty a daty neuniformními. Kromě typu dat si lze vybrat, podle jaké funkce bude generována d + 1. souřadnice. Tato souřadnice nemá vliv na průběh algoritmu, ovlivňuje pouze případnou vizualizaci vytvořené triangulace. Například ve dvourozměrném případě se zkonstruuje triangulace na základě souřadnic *x* a *y*, ale při vizualizaci se nezobrazí rovina, ale terén (obrázek dvou vrcholů z úvodního dialogu pochází z výpočtu náhodně vygenerované množiny uniformních dat, kde třetí souřadnice odpovídá součtu několika exponenciálních funkcí).

- Triangulation options – je tvořena několika volbami. Uživatel může určit, zda si přeje verifikovat vzniklou triangulaci (tato operace zejména ve 3D a pro větší datové množiny trvá dlouho), tj. všechny simplexy jsou otestovány, zda splňují princip prázdné kružnice (resp. koule) opsané. Dále lze specifikovat, jestli se statistické údaje mají zobrazovat a/nebo uložit do souboru na disk a zda se výsledná triangulace má uložit.
- Paralelization options – nabízí uživateli různé paralelní metody, které budou pro výpočet použity, různé způsoby distribuce mezi jednotlivá vlákna a počet vláken, která se použijí (je nesmyslem zvolit více vláken než má stroj procesorů, nejvyšší možná hodnota je 16 vláken). Celkem se uživateli nabízí 8 způsobů distribuce, každý způsob je identifikován celým číslem, přičemž způsoby se sudým číslem se nedoporučují, jedná se totiž o způsoby, které vstupní množinu bodů nerandomizují, a proto jsou-li body v množině uspořádané, celkový čas prudce narůstá.

Způsob	Význam
Equalizated raw data (0)	Vektor pořadí vkládání bodů tvoří uspořádanou posloupnost čísel 0..N-1. Tento vektor je rozdělen k-1 řezy na rovnoměrné intervaly, každý z těchto intervalů je přidělen jednomu vláknu.
Equalizated randomized data (1) = EQ1	Vektor pořadí vkládání bodů tvoří náhodně uspořádanou posloupnost čísel 0..N-1. Tento vektor je zpracován stejným způsobem jako v předchozím případě.
Equalized X-sorted data (2)	Vektor pořadí vkládání bodů tvoří posloupnost čísel 0..N-1 takovou, že pro i-tý prvek a_i této posloupnosti platí, že $F(a_{i-1}) < F(a_i) < F(a_{i+1})$, kde F(j) je funkce vracející *x*-souřadnici bodu, který je v množině vstupních bodů na j-tém místě. Tento vektor je zpracován opět stejným způsobem.
Randomized equalized X-sorted data (3) = EQ3	Jako předchozí metoda, ale pořadí vkládání v jednotlivých intervalech je vláknem před zahájení operace náhodně permutováno (sekundární randomizování).
Equalized XY-sorted data (4)	Obdoba způsobu *Equalized X-sorted data*, funkce $F(a_i)$ vrací druhou mocninu vzdálenosti a_i-tého bodu od počátku v rovině.
Randomized equalized XY-sorted data (5) = EQ5	Analogicky s *Randomized equalized X-sorted data*
Equalized XYZ-sorted data (6)	Analogicky s *Equalized XY-sorted data*, je uvažována třetí souřadnice (způsob má význam jen pro 3D verzi)
Randomized equalized XYZ-sorted data (7) = EQ7	Analogicky s *Randomized equalized XY-sorted data*

Co se týče volby metody použité pro výpočet, nabízí aplikace uživateli sériový způsob výpočtu (metoda je identifikována číslem 0), pesimistickou metodu (1), optimistickou s detekcí (2) a standardní zlodějskou (4). Ve dvourozměrném případě ještě metodu kružnice opsané (7). Ostatní metody, tj. optimistická s prevencí (3), rozšířená zlodějská (jen 3D, číslo metody je 5), dávková metodu (jen 3D, číslo metody je 6) a metoda koule opsané (taktéž číslo 7) nejsou obsaženy v základní binární podobě. Uživatel by neměl proto používat tyto metody, pokud nechce experimentovat, pak doporučena je pouze metoda s číslem 2.

Výpočet je zahájen okamžitě po stisku tlačítka OK umístěného v dolní části dialogu. Naopak stiskem tlačítka Cancel se všechna nastavení zruší a řízení se vrátí do výchozího dialogu.



Obr. B.4: *Dialog pro samostatný výpočet.*

Dávkové zpracování úlohy

Dávkové zpracování je řízeno jednoduchým textovým skriptem, který uživatel napíše v editoru skriptu. Každý příkaz skriptu je umístěn na nové řádce, prázdné řádky, komentáře atd. nejsou podporovány. Příkaz sestává z šesti parametrů oddělovaných čárkou, případné mezery či tabulátory jsou absorbovány. Parametrem může být buď jedna konstanta nebo pole konstant. Pole se zapisuje do hranatých závorek, jednotlivé položky jsou oddělené čárkou. Pokud si přejete otestovat data pro úplně všechny přípustné možnosti nějakého parametru, je možné namísto pole konstant použít zástupný symbol $.

Parametr	Možné hodnoty a význam parametru
1. Typ dat	Jedno z velkých písmen R,U,N či A. První tři jsou používané pro náhodně generovaná data **R**egulární, **U**niformní nebo **N**euniformní, následující parametr specifikuje velikost datové množiny tohoto typu. **A**dresář lze použít pro zpracování externích dat, následující parametr je vždy konstantní řetězec specifikující umístění souboru, který se má zpracovat, nebo adresáře, ze kterého budou zpracovány všechny soubory v něm se nacházející. Zástupný znak $ má význam [R,U,N].
2. N	Velikost datové množiny nebo cesta a jméno souboru nebo cesta adresáře.
3. Počet měření	Určuje, kolikrát bude každé měření provedeno (1 = bez opakování).
4. Počet vláken	Určuje počet použitých vláken pro paralelní metody, pro sekvenční algoritmus je tento parametr ignorován a proběhne vždy pouze jedno měření. Zástupný znak $ má význam [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16].
5. Způsob distribuce	Platné jsou hodnoty 0 až 7 identifikující způsob distribuce vstupní množiny mezi jednotlivá vlákna. Zástupný znak $ má význam [0, 1, 2, 3, 4, 5, 6, 7] a není doporučeno ho používat. Hodnoty vyšší jak 1 nejsou použitelné pro sekvenční algoritmus, a proto sudé hodnoty jsou automaticky transformovány na způsob s číslem 0, liché na 1. Pokud již sekvenční výpočet pro tento způsob distribuce proběhl, výpočet je přeskočen.
6. Metoda	Identifikační číslo metody (0 – sekvenční, 1 – pesimistická, …)

Příklady skriptů:

Skript 1 – tento skript ověřuje chování zlodějské metody v závislosti na distribuci vstupní množiny. Zkoumány jsou všechny dostupné distribuce (*EQ1*, *EQ3* a *EQ5).* Testována jsou jak reálná data (pro ně se testování neopakuje), tak i data uměle generována (test se opakuje čtyřikrát). Nejprve zpracuje všechny soubory umístěné v adresáři c:\dt\data, poté uměle vygenerovaná uniformní data s počtem bodů 5000, 10 tisíc …250 tisíc. Následuje test dalších externích dat a test uměle vytvořených pravidelných dat (tj. data typu grid).
A,c:\dt\data,1,2,[1,3,5],[4] U,[5000,10000,25000,50000,100000,150000,250000],5,2,[1,3,5],[4] A,c:\dt\data2,1,2,[1,3,5],[4] R,[4900,10000,24964, 50176,99856,149769,250000],5,2,[1,3,5],[4]
Skript 2 – tento skript testuje různá externí dat. První množina dat je zpracována sekvenčním algoritmem a pesimistickou metodou (závěrem tohoto testování tedy může být dosažené urychlení pesimistické metody), dále se zpracovává pouze samostatný soubor uni100tis.dat, tentokrát jen sekvenční metodou. Poslední „vyčerpávající" test ověřuje chování pesimistické, optimistické i zlodějské metody pro 2, 4, 6 a osm vláken. Veškeré testy používají distribuci *EQ1* a nejsou opakovány.
A,c:\dt\data,1,2,1,[0,1] A,c:\dt\material\uni100tis.dat,1,1,1,0 A,c:\dt\new_data,1,[2,4,6,8],1,[0,1,2,4]
Skript 3 – tento skript ukazuje použití zástupného znaku $. První příkaz skriptu provede úplné otestování reálných dat z adresáře c:\dt3d\real_data. Měření bude provedeno třikrát pro všechny možné distribuce, pro všechny paralelní metody a sekvenční metodu s nejrůznějším počtem vláknem. Každý soubor bude tedy zpracován 63x. Závěrem takového testování je urychlení jednotlivých paralelních metod v závislosti na počtu procesorů a v závislosti na způsobu distribuce vstupní množiny. Další příkaz se liší pouze ve změně množiny zpracovávaných dat, tentokrát se jedná o regulární, uniformní a neuniformní uměle vygenerovaná data. Poslední příkaz je užitečný pro ověření optimistické metody, nebudeme ho více rozebírat.
A,c:\dt3d\real_data,3,[$],[1,3,5,7],[0,1,2,3,4,5] [$],[1000,10000,25000,50000,100000,150000,250000,350000,500000],3,[$],[1,3,5,7],[0,1,2,3,4,5] A,c:\dt3d\data,1,[2,4,6,8],1,[0,2]

Zpracování skriptu je spuštěno po stisku tlačítka Run. Vytvářené triangulace jsou automaticky verifikovány a poté zahazovány, tj. nejsou ukládány na disk. Naopak na disk jsou ukládány veškeré statistiky o průběhu testu, a to do jediného textového souboru *Timings.log*. Každá nová statistika je zapsána na konec tohoto souboru. Kromě tohoto souboru vytváří interpret skriptu soubor *script.dts.triscr.log*, do kterého zapisuje právě zpracovávaný test, hvězdičkami je značen počet již uskutečněných opakování testu. Pokud během testu došlo k chybě, např. kvůli nedostatku paměti, zapíše se tato skutečnost do logu a test se s jinou randomizací opakuje. Jeden test se zkouší opakovat nejvýše třikrát, poté se prohlásí selhání a pokračuje se s dalším testem.

Statistika výpočtu

Výsledkem úspěšného výpočtu triangulace jsou výstupy dvojího typu. Jednak výstupem je triangulace uložená v paměti počítače, resp. výpis této triangulace do souboru, jednak se jedná o statistické informace, které rovněž mohou být uloženy na disk.

Statistické informace vypovídají o schopnosti paralelizovat úlohu a zahrnují údaje: jak dlouho trvaly jednotlivé fáze konstrukce *DT*, kolik času strávila jednotlivá vlákna v kritické sekci, čekala na uvolnění kritické sekce, kolik času bylo v průměru potřeba pro vkládání bodů jedním vláknem, počet uskutečněných operací rollback, počet potenciálních chyb zlodějské metody a další. Veškeré časové údaje jsou měřeny pomocí High-resolution timer (WINAPI funkce ::QueryPerformanceCounter a ::QueryPerformanceFrequency). Tímto způsobem je možné změřit časové intervaly kratší než 1ms s blíže nespecifikovanou chybou.

Jsou-li informace ukládány na disk, děje se tak vždy do aktuálního adresáře do souborů *Timings.X_Y_Z.log*, kde *X* je rok měření, *Y* den v roce a *Z* je počet milisekund uplynulých od restartování počítače zapsaný v hexadecimální soustavě. V případě dávkového zpracování se statistika ukládá vždy na konec souboru *Timings.Log*. Struktura statistiky je jednoduchá kvůli automatickému zpracování výsledků měření, které provádí utilita *Sumarizer.exe*. Jedná se o textový záznam, na každém řádku v souboru se nachází jeden údaj; časové údaje jsou uvedeny v milisekundách. Struktura je následující:

Řádek	Význam
1	Počet vkládaných bodů
2	Typ dat: 0 = regulární mříž, 1 = uniformní data, 2 = neuniformní; v případě čtení dat ze souboru tak je tato hodnota vyplněna podle přípony souboru: 0 = .REG, 1 = .UNI, 2 = .NON pro všechny ostatní přípony nelze automaticky rozhodnout a hodnota na tomto řádku uložená je –1
3	Pokud je na 2. řádku hodnota –1, pak na tomto řádku je jméno souboru, ze kterého byla data načtena, jinak je řádek prázdný
4	Počet vláken
5	Identifikační číslo použité metoda rozdělení dat mezi jednotlivá vlákna
6	Celková doba potřebná na *DT* (včetně randomizace pořadí vkládání, připravení dat na výstup a uvolnění *DAG* struktury)
7	Doba potřebná pro funkci DTLib::Initialize_DTProc
8	Doba potřebná pro spuštění vláken a vložení všech bodů vlákny, spočtená jako součet dílčích interních časů
9	Doba potřebná pro funkci DTLIb::Finalize_DTProc
10	Průměrná doba vkládání bodů na jedno vlákno
11	Průměrná doba na jedno vlákno, která byla strávena v kritické sekci (pouze pesimistická metoda, v ostatních případech je hodnota neplatná)
12	Průměrná doba na jedno vlákno, která byla strávena čekáním na vstup do kritické sekce (pouze pesimistická metoda, v ostatních případech je hodnota neplatná)
13	Identifikační číslo použité metody (0 – sekvenční, …)
14	počet simplexů nesplňujících Delaunayovo kritérium;-1 pokud výsledná triangulace nebyla verifikována
15	počet simplexů s chybnou orientací;-1 pokud výsledná triangulace nebyla verifikována
16	počet detekovaných deadlocků (pouze metody 2, 3, 4, 5)
17	Doba potřebná pro rozdělení množiny mezi vlákna, tj. čas spotřebovaný mediánem
18	Počet potenciálních problémů (pouze metody 4, 5)
19	Doba potřebná pro spuštění vláken a vložení všech bodů vlákny (naměřená)
20	Počet všech uzlů v *DAG* struktuře

Struktura vstupních a výstupních dat

Vstupní soubor se souřadnicemi bodů je textovým souborem, na jehož první řádce se nachází údaj o počtu bodů, které obsahuje. Souřadnice každého bodu jsou uloženy na samostatné řádce a tvoří je dvojice nebo trojice čísel oddělených jednou mezerou. Pokud se v souboru za blokem souřadnic nacházejí ještě další údaje, jsou ignorovány.

Výstupní soubor s *2D DT* obsahuje na svém začátku tatáž data, jako vstupní soubor. Za blokem souřadnic se nachází na samostatné řádce počet trojúhelníků a za ní blok vrcholů trojúhelníků. Každý trojúhelník je uložen na jedné řádce a sestává z trojice indexů popisujících vrcholy trojúhelníka.

Výstupní soubor s *3D DT* má sice stejnou strukturu, ale kvůli stávajícímu prohlížeči obsahuje navíc některé řádky, a proto zatímco výstupní soubor z dvourozměrné verze lze použít jako vstupní, u trojrozměrné verze to není možné. Příklad:

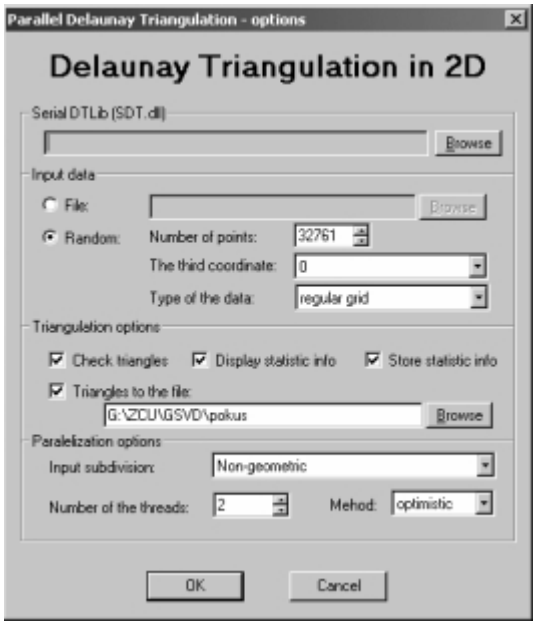
```
<new_net>
<vertex_3d>
9
0.42857 0.23810 0.47619
0.33333 0.04762 0.61905
0.52381 0.00000 0.19048
-0.00000 -0.20000 -0.00000
0.00000 0.00000 1.20000
0.19048 0.04762 0.90476
0.09524 0.00000 0.47619
0.00000 0.00000 0.57143
0.38095 0.00000 0.95238
<tetrahedron_by_vertices>
5
  3   4   7   8
  4   5   7   8
  5   6   7   8
  6   3   7   8
  1   2   7   8
```

Příloha C

Uživatelská dokumentace (MVE modul)

Rozdíly oproti aplikaci popsané v příloze B

Zásadní změnou je nemožnost dávkového zpracování, uskutečnit lze jen samostatný výpočet. Dialog pro nastavení a spuštění výpočtu doznal několika úprav. Předně dialog slouží pouze pro nastavení výpočtu, tj. jedná se o tzv. setup dialog. Výpočet je zahájen teprve po stisku příslušného tlačítka v MVE editoru. Protože se jedná o jediný způsob, jak nakonfigurovat modul, přibyla sekce Serial DTLib (viz obr. C.1), ve které uživatel specifikuje umístění knihovny se sekvenční verzí algoritmu (místo tlačítka setup v úvodním dialogu). Počet různých způsobů distribuce množiny vstupních bodů mezi jednotlivá vlákna byl redukován, stejně jako počet různých paralelních metod.



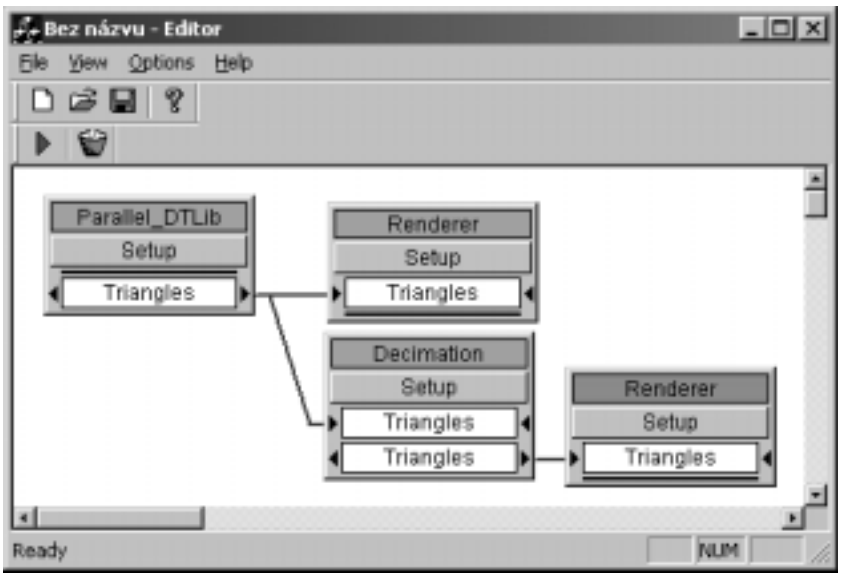
Obr. C.1: *Setup dialog*

Rovněž se změnila i struktura souborů se statistikou. Soubor je i nadále textový, ale neobsahuje všechny hodnoty a kromě pouhých hodnot zahrnuje i textové popisy. Příklad:

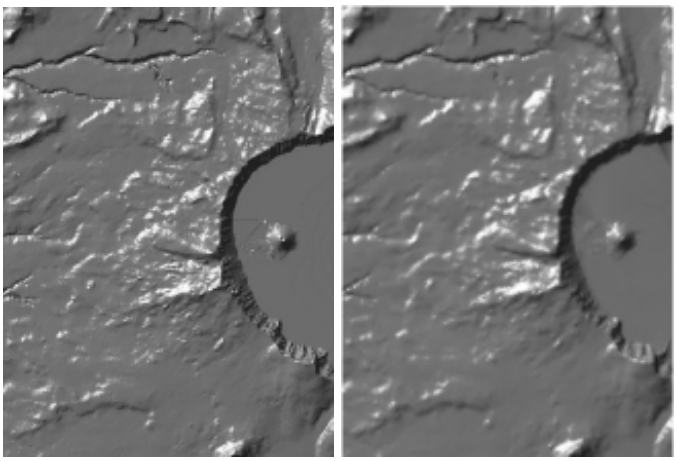
```
N = 32761
Distribution = REGULAR
Threads = 2
Input subdivision = XY-COORD
Total time [ms] = 5920.35093
Init time [ms] = 53.62367
Insert time [ms] = 4265.78201
Finish time [ms] = 1269.76099
Alg. time per thread [ms] = 3623.38938
Parallel method = OPTIMISTIC
Triangles without Delaunay property = 0
Triangles with wrong orientation = 0
Detected deadlocks = 0
```

Příklad použití modulu

V tomto nejjednodušším příkladě použijeme modul pouze pro vizualizaci (nicméně modul má širší uplatnění). Připojíme ho přímo k vizualizačnímu modulu a k modulu, který provádí decimování vytvořených triangulací, tj. redukuje velikost výstupních dat. Druhý vizualizační modul připojíme k tomuto decimačnímu modulu. Zapojení je uvedeno na obrázku C.2, příklad zobrazené triangulace pak na obrázku C.3.



Obr. C.2: *Vizualizace Delaunayovy triangulace*



Obr. C.3: *Model „Crater Lake“. Triangulace vlevo obsahuje 199144 trojúhelníků a 100001 bodů, na 70% decimovaná triangulace vpravo obsahuje pouhých 53984 trojúhelníků a 27434 bodů, nicméně rozdíl není příliš patrný. Data byla získána z kolekce M. Garland: Sample data for terrain simplification, http://graphics.cs.uiuc.edu/~garland/research/quadrics.html*

Evidenční list

Souhlasím s tím, aby moje diplomová práce byla půjčována k prezenčnímu studiu v Univerzitní knihovně ZČU v Plzni.

Datum: Podpis:

Uživatel stvrzuje svým podpisem, že tuto diplomovou práci použil ke studijním účelům a prohlašuje, že ji uvede mezi použitými prameny.

Jméno	Fakulka/katedra	Datum	Podpis