23

Paralelno programiranje

U ovom poglavlju razmatramo API-je i konstrukte za višenitno programiranje pomoću kojih možete iskoristiti procesor s više jezgara:

- Paralelni LINQ ili *PLINQ*
- Klasa `Parallel`
- Konstrukti za paralelno izvršavanje poslova
- Kolekcije koje podržavaju istovremenost

Navedene mogućnosti su uvedene s Frameworkom 4.0 a poznate su pod zajedničkim (veoma opštim) imenom PFX (Parallel Framework). Klasa `Parallel`, zajedno s konstruktima za paralelno izvršavanje poslova, zove se *Task Parallel Library* (biblioteka za paralelno izvršavanje poslova) ili TPL.

Pre čitanja ovog poglavlja potrebno je da dobro shvatite osnove opisane u poglavlju 14 – pre svega blokade, bezbedan višenitni rad i klasu `Task`.

Čemu PFX?

Relativno nedavno, proizvođači procesora su prešli sa jednog jezgra, na višejezgarne procesore. To je nama programerima prouzrokovalo problem jer naš standardni kôd napisan za rad u jednoprocesorskom okruženju neće automatski raditi brže samo zbog postojanja tih dodatnih jezgara.

Prednosti postojanja više jezgara mogu se na jednostavan način iskoristiti u većini serverskih aplikacija, gde svaka nit može nezavisno od drugih niti preuzeti obradu klijentskog zahteva, ali to je teže u slučaju klijentskih aplikacija – zato što onda svoj kôd koji teže opterećuje procesor morate najčešće prepraviti na sledeći način:

1. *Raspodelite* ga u manje celine.
2. Te celine izvršavajte paralelno primenom višenitnih tehnika.
3. Rezultate delimičnih obrada (međurezultate) *spajajte* čim postanu raspoloživi, na način koji je bezbedan u višenitnom okruženju i koji omogućava dobre performanse.

Mada sve navedeno možete postići i pomoću klasičnih konstrukata za višenitni rad, postupak je nezgrapan – naročito oni njegovi delovi koji se tiču raspodele obrade i spajanja međurezultata. Dodatan problem je to što uobičajena strategija postavljanja blokada radi bezbednog višenitnog rada prouzrokuje mnogo sukoba kada veći broj niti radi sa istim podacima u isto vreme.

PFX biblioteke su projektovane kako bi olakšale rešavanje problema u takvim slučajevima.

Programiranje s ciljem da se iskoristre prednosti postojanja više procesorskih jezgara ili više procesora, zove se *paralelno programiranje*. To je podskup šire teme višenitnog programiranja.

PFX koncepti

Postoje dve strategije za raspoređivanje opterećanja između više niti: *paralelizam podataka* (engl. *data parallelism*) i *paralelizam poslova* (engl. *task parallelism*).

Kada određen skup poslova treba izvršiti s velikim brojem vrednosti podataka, to možemo paralelizovati tako što svaka nit obavlja (isti) skup poslova s drugim podskupom izvornih vrednosti. To se zove *paralelizam podataka* zato što između niti raspodeljujemo *podatke*. Nasuprot tome, pri *paralelizmu poslova* raspodeljujemo *poslove*; drugim rečima, svaka nit obavlja različit posao.

Paralelizam podataka je uglavnom jednostavniji i bolje se skalira na hardveru s visokim stepenom paralelizma, zato što umanjuje ili potpuno uklanja dupliranje podataka (što smanjuje broj sukoba i probleme bezbednog izvršavanja niti). Osim toga, paralelizam podataka iskorišćava činjenicu da je broj vrednosti podataka znatno veći od broja različitih poslova, što povećava potencijal za paralelizam.

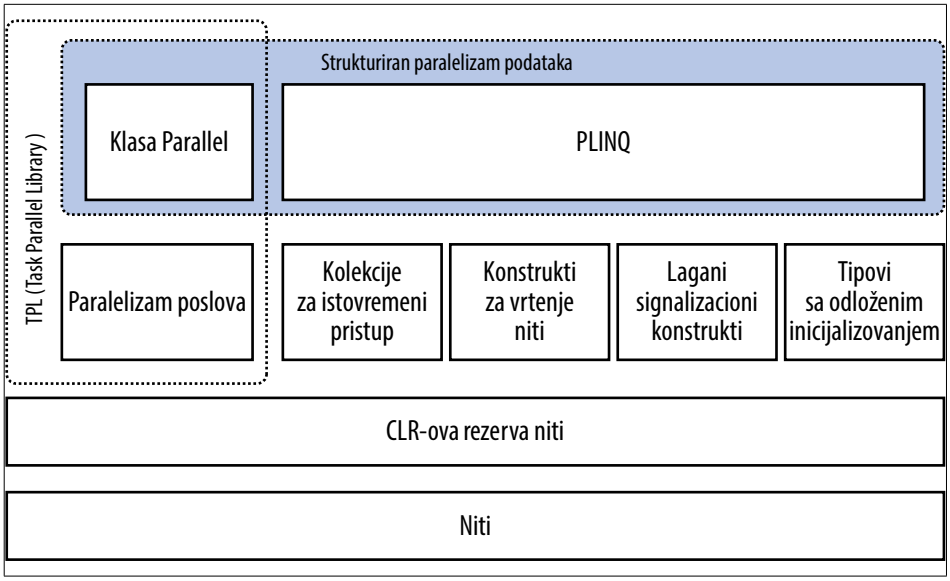
Paralelizam podataka dovodi i do *strukturiranog paralelizma*, što znači da jedinice paralelne obrade počinju i završavaju se na istim mestima u programu. Nasuprot tome, paralelizam poslova češće je nestrukturiran, što znači da jedinice paralelne obrade mogu počinjati i završavati se na nasumičnim mestima širom programa. Strukturiran paralelizam je jednostavniji, manje podložan greškama i omogućava da složenije poslove raspodele i koordinisanja niti (pa čak i spajanja međurezultata) grupišete i prepustite specijalizovanim bibliotekama.

PFX komponente

PFX se sastoji od dva sloja funkcionalnosti, kao što se vidi na slici 23-1. Gornji sloj čine dva API-ja za *strukturiran paralelizam podataka*: PLINQ i klasa `Parallel`. Donji sloj čine klase za paralelizam poslova – zajedno sa skupom dodatnih konstrukata koji olakšavaju aktivnosti u vezi s paralelnim programiranjem.

PLINQ pruža najbogatiju funkcionalnost: automatizuje sve korake postupka paralelizovanja obrade – uključujući i raspodelu obrade na poslove, izvršavanje tih poslova u nitima i spajanje međurezultata u jednu izlaznu sekvencu. To se zove *deklarativno* paralelizovanje – zato što samo deklarišete da želite paralelizovanje obrade (što strukturirate u obliku LINQ upita), a Frameworku prepuštate detalje izvedbe. Nasuprot tome, drugi pristupi su *imperativni* jer morate sami napisati dodatan kôd za raspodelu poslova ili spajanje međurezultata. Kada koristite klasu `Parallel`, morate sami spajati međurezultate; kada koristite konstrukte za paralelizam poslova, te poslove morate sami i raspodeliti:

	Raspodeljuje obradu	Spaja međurezultate
PLINQ	Da	Da
Klasa `Parallel`	Da	Ne
PFX-ov paralelizam poslova	Ne	Ne



Slika 23-1. PFX komponente.

Kolekcije za istovremeni pristup i konstrukti za vrtenje niti olakšavaju programiranje aktivnosti nižeg nivoa za paralelan rad. To su važni elementi zato što je PFX projektovan tako da radi ne samo na današnjem hardveru, nego i na budućim generacijama procesora koje će imati znatno veći broj jezgara. Ako želite da premestite gomilu iscepanih drva i to treba da obavi 32 radnika, najteži deo posla je da pri premeštanju drva radnici ne ometaju jedan drugog. Isti je slučaj i s raspodelom izvršavanja algoritma između 32 jezgra: ako pristup zajedničkim resursima štitite pomoću običnih blokada, rezultat može biti da samo deo svih tih jezgara bude zaista uposlen u isto vreme. Kolekcije za istovremeni pristup optimizovane su tako da omogućavaju vrlo veliki broj istovremenih pristupa, pri čemu nastoje da minimizuju ili eliminišu blokiranje niti. PLINQ i klasa `Parallel` koriste mogućnosti kolekcija za istovremeni pristup i konstrukata za vrtenje niti da bi efikasno upravljali obradom.

Druge upotrebe PFX-a

Konstrukti za paralelno programiranje korisni su ne samo kada imate više jezgara, nego i u drugim slučajevima:

- Kolekcije za istovremeni pristup ponekad su pogodne kad želite red čekanja, stek ili rečnik koji je bezbedan za višenitni rad.
- Klasa `BlockingCollection` pruža jednostavan način za implementiranje struktura tipa proizvođač/potrošač i dobar je način za *ograničavanje* broja istovremenih pristupa.
- Poslovi (objekti tipa `Task`) predstavljaju osnovu asinhronog programiranja, kao što smo videli u poglavlju 14.

Kada se koristi PFX

Primarni slučaj za upotrebu PFX-a jeste *paralelno programiranje*: iskorišćavanje mogućnosti višejezgarnih procesora radi bržeg izvršavanja koda koji teško opterećuje sistem.

Pri upošljavanju više jezgara, problem je Amdalov (Amdahl) zakon, prema kojem maksimalno poboljšanje performansi primenom paralelizovanja zavisi od dela koda koji mora da se izvršava sekvencijalno. Na primer, ako se može paralelizovati samo dve trećine vremena izvršavanja određenog algoritma, poboljšanje performansi ne može nikad biti veće od tri puta – čak i s beskonačnim brojem jezgara.

Stoga, pre nego što nastavite, vredi ispitati da li je usko grlo baš deo koda koji se može paralelizovati. Vredi proveriti i da li vaš kôd baš *mora* da obavlja toliki broj proračuna – optimizovanje koda je često najlakše i najefikasnije rešenje. Međutim, „cena“ nekih tehnika optimizovanja koda može biti teže paralelizovanje koda.

Dobici se najlakše postižu u slučaju onog što se ponekad naziva *ponižavajuće paralelni* (engl. *embarrassingly parallel*) problemi – gde se obrada lako može podeliti na poslove koji se samostalno efikasno izvršavaju (strukturiran paralelizam je veoma pogodan za tu vrstu problema). Primeri toga su poslovi obrade slika, iscrtavanja putanja svetlosnih zraka i rešavanja matematičkih ili kriptografskih problema primenom metode grube sile. Primer neponižavajuće paralelnog problema jeste implementiranje optimizovane verzije algoritma quicksort – za dobar rezultat potrebno je razmišljanje, a može zahtevati i nestrukturiran paralelizam.

PLINQ

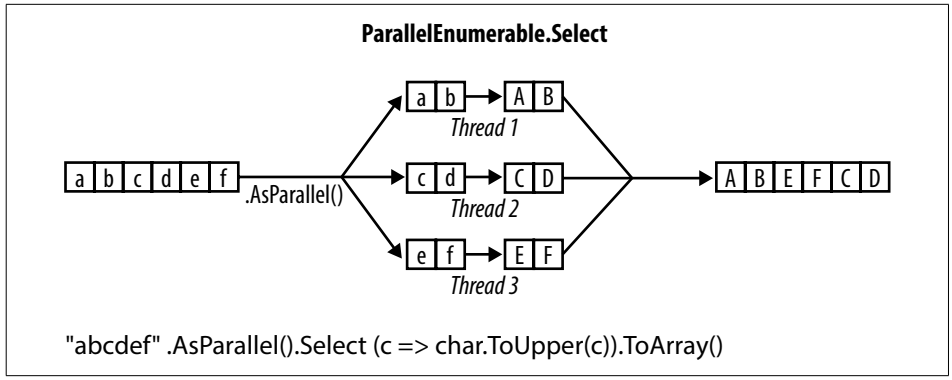
PLINQ automatski paralelizuje lokalne LINQ upite. PLINQ pruža prednost jednostavne upotrebe jer posao raspodele obrade i spajanja rezultata prebacuje na Framework.

Da biste iskoristili PLINQ, samo dodajte izlaznoj sekvenci `AsParallel()` a zatim nastavite LINQ upit na uobičajen način. Sledeći upit izračunava proste brojeve u opsegu između 3 i 100.000 – pri čemu upošljava sva jezgra na ciljnoj mašini:

```
// Izračunava proste brojeve pomoću jednostavnog (neoptimizovanog) algoritma.

IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);

var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;


int[] primes = parallelQuery.ToArray();
```

`AsParallel` je proširena metoda klase `System.Linq.ParallelEnumerable`. Ta metoda „umeće“ izvornu ulaznu sekvencu u sekvencu tipa `ParallelQuery<TSource>`, koja čini da se LINQ operatori koje zatim pozivate preslikavaju u alternativni skup proširenih metoda definisanih u klasi `ParallelEnumerable`. Te metode su paralelne implementacije svih standardnih operatora za upite, a rade tako što u suštini dele ulaznu sekvencu na delove koji se obrađuju u različitim nitima, a rezultate spajaju u jednu rezultujuću sekvencu za dalju obradu (slika 23-2).

Pozivanjem metode `AsSequential()`, sekvenca tipa `ParallelQuery` se „raspakiva“ u običnu sekvencu da bi se operatori za upite koje zatim pozivate razrešili u standardne operatore za upite i izvršili sekvencijalno. To je potrebno pre pozivanja metoda koje imaju sporedne efekte ili koje nisu bezbedne za višenitni rad.



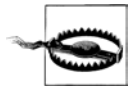
Slika 23-2. PLINQ-ov model izvršavanja.

U slučaju operatora za upite koji prihvataju dve ulazne sekvence (`Join`, `GroupJoin`, `Concat`, `Union`, `Intersect`, `Except` i `Zip`), metodu `AsParallel()` morate primeniti na obe ulazne sekvence (inače ćete izazvati izuzetak). Međutim, nema potrebe da metodu `AsParallel` neprekidno pozivate u upitu dok napreduje njegovo izvršavanje, zato što PLINQ-ovi operatori za upite proizvode druge izlazne sekvence tipa `ParallelQuery`. U stvari, višekratno pozivanje metode `AsParallel` ponovo uvodi neefikasnost jer izaziva spajanje i ponovno deljenje upita:

```
mySequence.AsParallel()              // Pretvara sekvencu u drugu
                                     // tipa ParallelQuery<int>
          .Where (n => n > 100)      // Vraća novu sekvencu ParallelQuery<int>
          .AsParallel()              // Nepotrebno – i neefikasno!
          .Select (n => n * n)
```


Ne mogu se svi operatori za upite efikasno paralelizovati. U slučaju onih za koje to nije moguće (videti odeljak „Ograničenja PLINQ-a“, na strani 831), PLINQ implementira operator sekvencijalno. PLINQ može raditi sekvencijalno i ako proceni da bi režijski deo postupka paralelizovanja zapravo usporio dati upit.

PLINQ je upotrebljiv samo za lokalne upite i ne radi za LINQ to SQL ili Entity Framework upite zato što se u tim slučajevima LINQ upit prevodi u SQL upit, koji se zatim izvršava na serveru baze podataka. Međutim, PLINQ *možete* iskoristiti za dodatno pretraživanje lokalnog skupa rezultata koji ste dobili pomoću upita u baze podataka.



Ako u PLINQ upitu dođe do izuzetka, on se ponovo generiše u obliku izuzetka tipa `AggregateException` čije svojstvo `InnerExceptions` sadrži stvarni izuzetak (ili izuzetke). Više informacija o tome naći ćete u odeljku „Izuzetak AggregateException“ na strani 854.

Zašto se metoda AsParallel ne poziva automatski?

Pošto se pozivanjem metode `AsParallel` LINQ upiti automatski paralelizuju, nameće se pitanje „Zašto Microsoft ne paralelizuje automatski standardne operatore za upite da bi PLINQ bio podrazumevani način obrade?“

To *opciono* rešenje je odabrano iz više razloga. Prvo, da bi PLINQ bio od koristi, treba da postoji razumna količina složene obrade koja bi se raspodelila na radne niti. Budući da se većina upita tipa LINQ to Objects izvršava vrlo brzo, paralelizovanje bi bilo ne samo nepotrebno, nego bi i režijski deo posla raspodele obrade, spajanja međurezultata i koordinisanja dodatnih niti zapravo usporio celu stvar.

Osim toga:

- Rezultat PLINQ upita (standardno) može biti različit od rezultata LINQ upita po pitanju redosleda elemenata (videti odeljak „PLINQ i redosled elemenata izlazne sekvence“, na ovoj strani).
- PLINQ pakuje sve izuzetke u izuzetak tipa `AggregateException` (da bi podržao mogućnost generisanja izuzetaka više vrsta).
- PLINQ vraća nepouzdane rezultate ako upit poziva metode koji nisu bezbedne za višenitni rad.

I najzad, PLINQ pruža priličan broj mogućnosti za fino podešavanje. Opterećivanje standardnog API-ja LINQ to Objects detaljima te vrste samo bi odvlačilo pažnju.

Postupak paralelnog izvršavanja upita

Isto kao i obični LINQ upiti, PLINQ upiti se izvršavaju odloženo. To znači da se izvršavanje pokreće samo kada vam zatreba rezultat upita – najčešće unutar petlje `foreach` (mada to može biti i u slučaju operatora za konverziju kao što je `ToArray` ili drugog operatora koji vraća pojedinačni element ili vrednost).

Međutim, dok nabrajate rezultate, PLINQ upiti se izvršavaju donekle drugačije od običnog sekvencijalnog upita. Sekvencijalni upit korisnik u potpunosti izvršava na „potezni“ način: svaki element ulazne sekvence povlači se i obrađuje tek kada to korisnik izričito zatraži. Paralelni upit obično koristi nezavisne niti za povlačenje elemenata iz ulazne sekvence neznatno *pre* nego što oni zatrebaju korisniku (slično aplikacijama za prikazivanje vesti koje prikazuju naslov sledeće vesti, ili slično baferu za sprečavanje preskakanja sadržaja u CD plejerima). Upit zatim paralelno obrađuje elemente u lancu, pri čemu rezultate čuva u malom baferu da bi bili spremni kada ih korisnik zahteva. Ako korisnik pauzira ili ranije prekine nabrajanje rezultata upita, obrada upita se takođe zaustavlja ili otkazuje da se ne bi bez potrebe tratilo vreme procesora ili memorija.

Ponašanje PLINQ-a možete finije podesiti ako iza metode `AsParallel` pozovete metodu `WithMergeOptions`. Podrazumevana vrednost parametra te metode, opcija `AutoBuffered`, uglavnom daje najbolje ukupne rezultate. Opcija `NotBuffered` isključuje bafer i korisna je ako želite da vidite rezultat čim to bude moguće; opcija `FullyBuffered` smešta ceo skup rezultata u keš pre nego što ga prosledi korisniku (operatori `OrderBy` i `Reverse` prirodno rade na taj način, što važi i za operatore nad pojedinačnim elementima i operatore za agregiranje i konverziju elemenata ulazne sekvence)

PLINQ i redosled elemenata izlazne sekvence

Sporedni efekat paralelizovanja operatora upita jeste sledeći: kada se spajaju međurezultati, redosled nije obavezno isti kao pri povlačenju ulaznih elemenata (slika 23-2). Drugim rečima, više ne važi LINQ-ova standardna garancija očuvanja redosleda elemenata u sekvencama.

Ako redosled treba da ostane sačuvan, iza metode `AsParallel()` pozovite metodu `AsOrdered()`:

```
myCollection.AsParallel().AsOrdered()...
```

Pozivanje metode `AsOrdered` dovodi do slabljenja performasi u slučaju sekvenci sa velikim brojem elemenata zato što PLINQ mora da pamti izvorni položaj svakog ulaznog elementa.

Efekat pozivanja metode `AsOrdered` možete poništiti kasnije u upitu ako pozovete metodu `AsUnordered`: to uvodi „tačku nasumičnog mešanja“ što omogućava da se upit efikasnije izvršava od tog mesta nadalje. Na primer, ako želite da sačuvate isti redosled rezultata kao elemenata ulazne sekvence, ali da to važi samo za prva dva operatora upita, uradite sledeće:

```
inputSequence.AsParallel().AsOrdered().QueryOperator1()
  .QueryOperator2()
  .AsUnordered()       // Odavde nadalje, redosled je nebitan
  .QueryOperator3()
  ...
```

`AsOrdered` nije podrazumevana opcija zato što je u većini upita izvorni redosled elemenata nevažan. Drugim rečima, kada bi `AsOrdered` bila podrazumevana opcija, u većini svojih paralelnih upita morali biste eksplicitno zadavati opciju `AsUnordered` da biste postigli najbolje performanse, što bi bilo nezgrapno.

Ograničenja PLINQ-a

Zasad postoji nekoliko praktičnih ograničenja toga šta sve PLINQ može da paralelizuje. Ta ograničenja će možda biti ublažena u budućim servisnim paketima i verzijama Frameworka.

Sledeći operatori upita onemogućavaju paralelizovanje upita, osim kada su elementi izvorne sekvence na svojim izvornim indeksnim položajima:

- `Take`, `TakeWhile`, `Skip` i `SkipWhile`
- Indeksirane verzije `Select`, `SelectMany` i `ElementAt`

Većina operatora upita menja indeksne položaje elemenata (što važi i za operatore koji uklanjaju elemente iz ulazne sekvence, kao što je operator `Where`). To znači sledeće: ako želite da koristite prethodne operatore, morate ih postaviti najčešće na početak upita.

Sledeći operatori upita mogu se paralelizovati, ali primenjuju „skupu“ strategiju raspodele obrade koja ponekad može biti sporija od sekvencijalnog načina obrade:

- `Join`, `GroupBy`, `GroupJoin`, `Distinct`, `Union`, `Intersect` i `Except`

Preklopljene verzije operatora za agregiranje elemenata sa *početnim zrnom*, u svojim standardnim varijantama ne mogu se paralelizovati – PLINQ ima specijalne verzije svojih operatora za te slučajeve (videti odeljak „Optimizovanje PLINQ-a“, na strani 835).

Svi ostali operatori se mogu paralelizovati, mada upotreba paralelizovanih operatora ne garantuje da će upit biti paralelizovan. PLINQ može izvršiti upit na sekvencijalan način ako proceni da bi režijski deo postupka paralelizovanja usporio dati upit. To ponašanje možete isključiti i nametnuti paralelizam ako iza metode `AsParallel()` zadate sledeće:

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

Primer: paralelni program za proveru pravopisne ispravnosti

Pretpostavimo da nam treba program za proveru engleskog pravopisa koji veoma brzo obrađuje obimne dokumente tako što upošljava sva procesorska jezgra s kojima raspolaže. Ako algoritam formulišemo u obliku LINQ upita, možemo ga vrlo lako paralelizovati.

Prvi korak je priprema rečnika engleskih reči u obliku objekta tipa `HashSet` radi efikasnog pretraživanja:

```
if (!File.Exists ("WordLookup.txt"))    // Sadrži približno 150.000 reči
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

Zatim pomoću našeg rečnika (objekta `wordLookup`) pravimo probni „dokument“ čiji je sadržaj niz od milion nasumično izabranih reči. Pošto formiramo taj niz, uvodimo nekoliko pravopisnih grešaka:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Uvodimo nekoliko
wordsToTest [23456] = "wubsie";     // pravopisnih grešaka.
```

Sada možemo paralelno ispitivati tačnost pravopisa tako što poredimo sadržaj niza `wordsToTest` sa sadržajem rečnika `wordLookup`. PLINQ čini taj postupak veoma jednostavnim:

```
var query = wordsToTest
  .AsParallel()
  .Select  ((word, index) => new IndexedWord { Word=word, Index=index })
  .Where   (iword => !wordLookup.Contains (iword.Word))
  .OrderBy (iword => iword.Index);

foreach (var mistake in query)
  Console.WriteLine (mistake.Word + " - index = " + mistake.Index);

// REZULTAT:
// woozsh - index = 12345
// wubsie - index = 23456
```

`IndexedWord` je namenska struktura definisana kao što sledi:

```
struct IndexedWord { public string Word; public int Index; }
```

Metoda `wordLookup.Contains` u predikatu dodaje upitu malo „mesa“ zbog kojeg ga vredi paralelizovati.

Upit bi bio nešto jednostavniji da smo upotrebili anonimni tip umesto strukture `IndexedWord`. Međutim, to bi pogoršalo performanse zato što je za anonimne tipove (pošto su to klase, to su referentni tipovi) potreban dodatan režijski rad dodeljivanja memorije na hipu a kasnije i sakupljanje smeća (kad ta memorija više ne bude potrebna).

Razlika možda nije velika u slučaju sekvencijalnih upita, ali kod paralelnih upita, rešenje s dodeljivanjem memorije na steku može može pružiti prilično veliku prednost. Razlog je to što se postupak dodeljivanja memorije na steku može u velikoj meri paralelizovati (zato što svaka nit ima vlastiti stek), dok se nasuprot tome sve niti takmiče za isti hip – kojim upravljaju isti upravljač memorijom i isti sakupljač smeća.

Upotreba klase ThreadLocal<T>

Naš primer možemo proširiti paralelizovanjem same liste nasumično izabranih probnih reči. Pošto smo njeno sastavljanje strukturirali u obliku LINQ upita, to bi trebalo da bude lako. Ovako izgleda sekvencijalna verzija upita:

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();
```

Nažalost, pozivanje `random.Next` nije bezbedno za višenitni rad, pa zato postupak nije tako jednostavan kao umetanje operatora `AsParallel()` u upit. Moguće rešenje je da napišemo funkciju koja postavlja blokadu oko `random.Next`; međutim, to ograničava mogućnost istovremenog pristupa iz više niti. Bolje rešenje je upotreba objekta tipa `ThreadLocal<Random>` (videti odeljak „Skladištenje podataka lokalno u nitima“, na strani 817, u prethodnom poglavlju) da bismo u svakoj niti napravili zaseban objekat `Random`. Upit možemo zatim paralelizovati na sledeći način:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );

string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
  .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
  .ToArray();
```

Našoj fabričkoj funkciji koja instancira objekat `Random`, prosleđujemo heš vrednosti `Guid` kako bismo obezbedili da ako se unutar vrlo kratkog razdoblja generišu dva `Random` objekta, oni proizvode različite sekvence nasumičnih brojeva.

Kada se može koristiti PLINQ

Možete pasti u iskušenje da u svojim postojećim aplikacijama prepravite LINQ upite i eksperimentišete s njihovim paralelizovanjem. To je najčešće pogrešno, zato što se većina problema za koje je LINQ očigledno najbolje rešenje, obično izvršava vrlo brzo i zato paralelizovanje ne bi ništa poboljšalo. Bolje rešenje je da pronađete uska grla koja teško opterećuju procesor i da u tim slučajevima razmislite „Može li se ovo izraziti u obliku LINQ upita?“ (Dobrodošao sporedni efekat takvog restrukturiranja jeste to što LINQ obično čini programski kôd sažetijim i razumljivijim.)

PLINQ je pogodan za ponižavajuće paralelne probleme. Međutim, to može biti loš izbor za obradu slika, zato što spajanje međurezultata obrade piksela u jednu izlaznu sekvencu formira usko grlo. Umesto toga, bolje je da piksele upisujete direktno u niz ili u blok neupravljane memorije i da višenitnom obradom upravljate pomoću klase `Parallel` ili pomoću paralelizma poslova. (Međutim, moguće je isključivanje spajanja delimičnih rezultata pomoću operatora `ForAll` – što razmatramo u odeljku „Optimizovanje PLINQ-a“, na strani 835. To je logičan postupak ako se algoritam za obradu slika prirodno može izraziti u obliku LINQ upita.)

Funkcionalna čistoća

Budući da PLINQ izvršava upit u više paralelnih niti, morate voditi računa o tome da ne pokrećete operacije koje nisu bezbedne za višenitni rad. Pre svega, upisivanje vrednosti u promenljive proizvodi *sporedne efekte* i zato nije bezbedno u višenitnom radu:

```
// Sledeći upit množi svaki element njegovim indeksom.
// Ako krenemo od opsega vrednosti tipa Enumerable.Range(0,999),
// trebalo bi da rezultat budu kvadrati tih vrednosti.
int i = 0;
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

Operaciju inkrementiranja tekuće vrednosti `i` mogli smo da načinimo bezbednom za višenitni rad pomoću blokada, ali bi nam i dalje ostao problem da vrednost `i` neće uvek odgovarati indeksu tekućeg ulaznog elementa koji se obrađuje. Dodavanje operatora `AsOrdered` upitu ne bi rešilo taj problem, zato što `AsOrdered` obezbeđuje da se obrađeni elementi slažu redosledom kojim bi bili kada bi se obrađivali sekvencijalno – ali ne garantuje da se oni *zaista* obrađuju sekvencijalno.

Umesto toga, ovaj upit treba napisati tako da koristi indeksiranu verziju operatora `Select`:

```
var query = Enumerable.Range(0,999).AsParallel().Select ((n, i) => n * i);
```

Za najbolje performanse, trebalo bi da svaka metoda koja se poziva iz operatora bude bezbedna za višenitni rad zato što je projektovana tako da ne upisuje ništa ni u polja, ni u svojstva (nema sporedne efekte, tj. metoda je *funkcionalno čista*). Ako je metoda bezbedna za višenitni rad zato što sadrži *blokade*, mogućnost paralelizovanja upita biće biće ograničena – na trajanje blokade podeljeno s ukupnim trajanjem izvršavanja te funkcije.

Podešavanje stepena paralelizma

PLINQ standardno sam određuje optimalan stepen za raspoloživ procesor. Možete zadati drugačije ako pozovete metodu `WithDegreeOfParallelism` iza `AsParallel`:

```
...AsParallel().WithDegreeOfPallelism(4)...
```

Primer slučaja kada biste stepen paralelizma možda povisili na vrednost veću od broja jezgara jeste pri ulazno/izlaznim operacijama (recimo, preuzimanje više veb stranica istovremeno). Međutim, u C# 5 i Frameworku 4.5, kombinatori poslova i asinhrone funkcije pružaju rešenje koje je slično po jednostavnosti i *efikasnosti* (videti odeljak „Kombinatori poslova“, na strani 541 poglavlja 14. Za razliku od objekata `Task`, PLINQ ne može da obavlja ulazno/izlazne operacije bez blokiranja niti (i to niti *iz rezerve*, da bi stvari bile još gore).

Menjanje stepena paralelizma

Metodu `WithDegreeOfParallelism` možete pozvati samo jedanput unutar PLINQ upita. Ako treba da je pozovete ponovo, morate pokrenuti spajanje rezultata i ponovnu raspodelu obrade u upitu tako što unutar upita ponovo pozovete `AsParallel()`:

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism (2)
  .Where (c => !char.IsWhiteSpace (c))
  .AsParallel().WithDegreeOfParallelism (3)   // Pokreće spajanje rezultata i novu
                                              // raspodelu obrade
  .Select (c => char.ToUpper (c))
```

Otkazivanje upita

Otkazivanje PLINQ upita čije rezultate preuzimate u petlji `foreach` lako je: samo izađite iz petlje `foreach` i izvršavanje upita se automatski otkazuje jer se enumerator implicitlno oslobađa.