PRESTANDA- OCH BETEENDEANALYS AV PARALLELLA KÖER MED ITERATOR

Examensarbete Systemarkitekturutbildningen

Viktor Lodin
Magnus Olovsson

VT 2014:KSAI02

UNIVERSITY OF BORÅS
SCHOOL OF BUSINESS AND IT

Systemarkitekturutbildningen är en kandidatutbildning med fokus på programutveckling. Utbildningen ger studenterna god bredd inom traditionell program- och systemutveckling, samt en spets mot modern utveckling för webben, mobila enheter och spel. Systemarkitekten blir en tekniskt skicklig och mycket bred programutvecklare. Typiska roller är därför programmerare och lösningsarkitekt. Styrkan hos utbildningen är främst bredden på de mjukvaruprojekt den färdige studenten är förberedd för. Efter examen skall systemarkitekter fungera dels som självständiga programutvecklare och dels som medarbetare i en större utvecklingsgrupp, vilket innebär förtrogenhet med olika arbetssätt inom programutveckling.

I utbildningen läggs stor vikt vid användning av de senaste teknikerna, miljöerna, verktygen och metoderna. Tillsammans med ovanstående teoretiska grund innebär detta att systemarkitekter skall vara anställningsbara som programutvecklare direkt efter examen. Det är lika naturligt för en nyutexaminerad systemarkitekt att arbeta som programutvecklare på ett stort företags IT-avdelning, som en konsultfirma. Systemarkitekten är också lämpad att arbeta inom teknik- och idédrivna verksamheter, vilka till exempel kan vara spelutveckling, webbapplikationer eller mobila tjänster.

Syftet med examensarbetet på systemarkitekturutbildningen är att studenten skall visa förmåga att delta i forsknings- eller utvecklingsarbete och därigenom bidra till kunskapsutvecklingen inom ämnet och avrapportera detta på ett vetenskapligt sätt. Således måste de projekt som utförs ha tillräcklig vetenskaplig och/eller innovativ höjd för att generera ny och generellt intressant kunskap.

Examensarbetet genomförs vanligen i samarbete med en extern uppdragsgivare eller forskningsgrupp. Det huvudsakliga resultatet utgörs av en skriftlig rapport på engelska eller svenska, samt eventuell produkt (t.ex. programvara eller rapport) levererad till extern uppdragsgivare. I examinationen ingår även presentation av arbetet, samt muntlig och skriftlig opposition på ett annat examensarbete vid ett examinationsseminarium. Examensarbetet bedöms och betygssätts baserat på delarna ovan, specifikt tas även hänsyn till kvaliteten på eventuell framtagen mjukvara. Examinator rådfrågar handledare och eventuell extern kontaktperson vid betygssättning.

HÖGSKOLAN I BORÅS

INSTITUTIONEN HANDELS- OCH IT-HÖGSKOLAN

BESÖKSADRESS: JÄRNVÄGSGATAN 5 · POSTADRESS: ALLÉGATAN 1, 501 90 BORÅS
TFN: 033-435 40 00 · E-POST: INST.HIT@HB.SE · WEBB: WWW.HB.SE/HIT

Svensk titel: Prestanda- och beteendeanalys av parallella köer med iterator.

Engelsk titel: Performance and behavior analysis of concurrent queues with iterator.

Utgivningsår: 2014

Författare: Viktor Lodin, Magnus Olovsson

Handledare: Anders Gidenstam

Abstract

In modern hardware development there is a big focus on producing processors with more and more cores. Due to this the software need to develop in the best possible way to take advantage of this parallel potential. A big part of this is to be able to share data between several parallel processes, which is achieved with the help of concurrent collection data types. A common operation in collection data types is iteration. The goal of the study was to analyze the performance and behavior of several known algorithms for iteration of the collection data type queue. Because different system parameters can alter the performance of the iterators, these has been evaluated as well. Examples of preconditions are the number of worker threads which works against the queue, different initial sizes of the queue and different pinning strategies. Initial size describes how many elements the queue holds at the beginning of the experiments and pinning strategies describe what core each thread should bind itself to. Some of the iterator algorithms can guarantee that the returned state is an atomic snapshot of the queue. An atomic snapshot is a snapshot of the queue at some fixed point in time. Because of this, a goal of this study was to evaluate the cost to get this guarantee. In addition to this, the performance of the enqueue and dequeue operations for each queue has been tested to get an overview of the queues performance.

To measure performance a micro benchmark application has been implemented. This benchmark provides an interface for all queues to implement and can through this interface test the performance of the queues. The micro benchmark measures the performance of the various operations of the queue. The way the queue has been pressured during these benchmarks is not realistic for how the queue might be used in real-life situations. Instead the performance is measured at highest possible load. This is done to ease the performance comparison between the queues and give a picture of a worst-case scenario.

In this study the performance of four queues with iterators has been tested. The experiments has been done in C# with .NET 4.5 within a Windows environment. The built in parallel queue in .NET was one of the tested queues. Partly due to that's its interesting how well Microsoft has optimized it, but also to have a base in the comparison between the other tested queues. Michael and Scott's queue is another of the queues tested. To this queue two different iterators has been added. These are Scan and Return and Double Collect. Also, a concurrent queue implemented from universal methods for constructing concurrent data objects from sequential ones and based on the immutable queue available in the .NET library has been tested.

The results from the performed benchmarks shows that the Michael and Scott queue with the Scan and Return iterator is the fastest at iteration. Michael and Scott with the Double Collect iterator is the second fastest at iteration. The fastest enqueue and dequeue operations is found within the parallel queue from the .NET library. The queue based on the immutable queue shows to be the slowest of the tested at both iterations and enqueue and dequeue operations. The cost of the guarantee for consistency is measured between the Scan and Return and Double Collect iterators. These two are chosen because these are the two fastest iterators as well as Scan and Return do not leave this guarantee while Double Collect do. This cost shows to be relatively large, Scan and Return performs up to three times as fast as Double Collect.

With the help of the results of this study, programmers can make well informed choices of which queue and iterator algorithm to choose to optimize their systems. This might be of most importance of development of larger systems, but can also be useful within smaller systems.

Keywords: Parallel programming, queue, iterators, C#, .NET, performance, data analysis

Sammanfattning

I modern utveckling av hårdvara ligger det stort fokus på att producera processorer med fler och fler kärnor. Därmed behöver även mjukvaran utvecklas för att på bästa sätt utnyttja all denna parallella potential. En stor del av detta är då att kunna dela data mellan flera parallella processer, vilket uppnås med hjälp av parallella samlingsdatatyper. En vanlig operation på samlingsdatatyper är att iterera denna. Studiens mål var att analysera prestanda och beteende hos ett flertal kända algoritmer för iteration av datasamlingen kö. Även hur olika förutsättningar kan påverka iteratorns prestanda har värderats. Några exempel på dessa förutsättningar är antalet arbetstrådar som arbetar mot kön, initial storlek hos kön samt olika pinning strategier. Initial storlek beskriver hur många element som befinner sig i kön vid experimentens start och pinning strategi beskriver vilken kärna varje tråd skall binda sig till. Vissa iterator algoritmer lämnar garantier för att det tillstånd som returneras är ett atomiskt snapshot av kön. Ett atomiskt snapshot är en ögonblicksbild av hur kön såg ut vid någon fast tidpunkt. På grund av detta har det även varit ett mål att mäta hur stor kostnaden är för att få denna garanti. Utöver detta har prestandan hos enqueue och dequeue operationerna för respektive kö testats för att få en helhetsblick över köns prestanda.

För att mäta prestandan har ett benchmarkprogram implementerats. Detta benchmarkprogram förser ett gränssnitt för samtliga köer att implementera, och kan utefter detta gränssnitt testa prestandan hos kön. Programmet kör mikrobenchmarks som mäter prestandan hos varje enskild operation hos kön. Det sätt som kön pressas på under dessa benchmarks är inte realistiskt för hur kön kan tänkas användas i skarpt läge. Istället mäts prestandan vid högsta möjliga belastning. Detta görs för att enklast kunna jämföra prestandan mellan de olika köerna.

I studien har prestandan hos fyra köer med iteratorer testats, experimenten är utförda i C# med .NET 4.5 i en Windows miljö. Den parallella kö som finns i .NET biblioteket var en av köerna som testades. Dels för att det är intressant att se hur väl Microsoft optimerat denna, men också för att få en utgångspunkt att jämföra med de andra testade köerna. Michael och Scotts kö har även den testats, med två stycken olika iteratorer tillagda. Dessa är Scan and Return och Double Collect. Även en parallell kö framtagen med hjälp av universella metoder för att konstruera paralllella dataobjekt från sekventiella, baserad på den immutable kö som finns i .NET biblioteket har testats. En immutable kö är en kö som inte kan modifieras efter initiering.

Resultaten från utförda benchmarks visar att Michael och Scott kön med Scan and Return iteratorn är den snabbaste på iteration, med Double Collect iteratorn som tvåa. Snabbast enqueue och dequeue operationer hittas i .NET bibliotekets parallella kö. Kön som bygger på immutable visar sig vara långsammast vad gäller iteration i de flesta fall. Den är även långsammast vad gäller enqueue och dequeue operationerna i samtliga fall. Kostnaden för att få en garanti för ett atomiskt snaphot mäter vi i skillnaden mellan Scan and Return och Double Collect iteratorerna. Detta på grund av att dessa är de två snabbaste iteratorerna och Scan and Return inte lämnar garantin medan Double Collect gör det. Denna kostnad visar sig vara relativt stor, Scan and Return presterar upp emot tre gånger så snabbt som Double Collect.

Med hjälp av resultaten från denna studie kan nu utvecklare göra väl informerade val vad gäller vilken kö med iterator algoritm de skall välja för att optimera sina system. Detta kanske är som viktigast vid utveckling av större system, men kan även vara användbart vid mindre.

Nyckelord: Parallell programmering, kö, iteratorer, C#, .NET, prestanda, data analys

Innehållsförteckning

1 Inledning - 1 -
1.1 Bakgrund - 1 -
1.2 Problem - 1 -
1.3 Frågeställning - 2 -
1.4 Avgränsningar - 2 -
2 Metod - 3 -
2.1 Mikrobenchmark - 3 -
2.1.1 Parametrar - 3 -
2.1.2 Pinning - 4 -
2.2 Insamling av data - 4 -
2.3 Analys av data - 5 -
3 Teori - 7 -
3.1 Parallella datastrukturer - 7 -
3.2 Algoritmer - 7 -
3.2.1 Michael och Scott - 8 -
3.2.2 .NET bibliotekets parallella kö - 10 -
3.2.3 Kön som bygger på Immutable - 11 -
3.3 Pinning - 12 -
3.4 Benchmarking - 13 -
3.4.1 Benchmarkprogrammet - 13 -
3.4.2 Arbetstråd - 13 -
3.4.3 Mätning - 13 -
3.4.4 Förutsättningar - 14 -
3.5 Hårdvara - 14 -
3.5.1 Processorns minneshierarki - 14 -
3.6 Relaterade arbeten - 14 -
4 Resultat - 16 -
4.1 Val av parametervärde - 16 -
4.1.1 Pinning - 16 -
4.2 Implementation - 18 -
4.2.1 Benchmark - 18 -
4.2.2 Michael och Scott kön - 20 -
4.2.3 Scan and Return - 20 -
4.2.4 Double Collect - 20 -
4.2.5 .NET Framework - 21 -
4.2.6 Kö som bygger på ImmutableQueue - 21 -
4.3 Experiment - 22 -
4.3.1 Experimentella förutsättningar - 22 -
4.3.2 Resultat från experiment - 22 -
4.3.3 Grafer - 22 -
5 Analys - 33 -
5.1 Pinning strategier - 33 -
5.1.1 None - 33 -
5.1.2 Alternate - 33 -
5.1.3 Fill - 34 -
5.1.4 Sammanfattning av pinning strategier - 34 -
5.2 Initial storlek - 35 -
5.3 Köer - 39 -
5.3.1 Framework - 39 -
5.3.2 Kö som bygger på Immutable - 39 -
5.3.3 Scan and Return och Double Collect - 40 -
5.3.4 Sammanfattning av köer - 40 -

5.4 Jämförelse med Java .... - 41 -
5.5 Sammanfattning av analys .... - 42 -
6 Slutsats .... - 45 -
6.1 Diskussion .... - 45 -
6.1.1 Reliabilitet .... - 45 -
6.1.2 Reproducerbarhet .... - 45 -
6.1.3 Generalitet .... - 46 -
6.1.4 Validitet .... - 46 -
6.2 Fortsatta studier .... - 47 -
Referenser .... - 48 -
Bilagor i
Bilaga A – Gloslista i
Bilaga B – Kod ii
Framework ii
Immutable iii
Michael och Scott iv
Scan and Return vi
Double Collect vii
Benchmark viii
Gränssnitt kö xvi

1 Inledning

1.1 Bakgrund

I dagens läge utvecklas inte hastigheten hos processorernas enskilda kärnor speciellt mycket (Fuller & Millett 2011). Istället ligger fokus på att bygga processorer som kan arbeta med fler uppgifter samtidigt. Framsteg uppnås både genom att processorn får fler kärnor samt att varje kärna kan utföra flera uppgifter samtidigt, så kallat hypertrådning.

Vid utveckling av prestandakritiska system har man länge varit van vid att kunna hantera en växande problemstorlek, eftersom processorernas kärnor tidigare har haft en ständig prestandaökning. Det har medfört att utvecklade system automatiskt har skalat med nyare processorer, det vill säga att en ny processor gav ett snabbare system. Detta kommer inte stämma längre nu när fokus inom processorutveckling ligger på att öka parallellismen snarare än att öka varje kärnas hastighet (Fuller & Millett 2011). För att kunna fortsätta hantera en växande problemstorlek måste nu mjukvaran anpassas för att hantera all parallellism som introduceras i hårdvaran. Helst skall systemen skala med hårdvarans parallellism. Ett av de största problemen som uppstår vid parallell programmering är att flera trådar vill kunna använda gemensam data. Det uppstår då många problem, som enklast löses med parallella datastrukturer. En operation som ofta finns i parallella datastrukturer är att iterera över den. Den finns till exempel i .NET's (Microsoft u.å. a) Java's (Oracle u.å.) och Intel TBB's (Intel u.å.) implementationer av parallella köer. I system där prestandan är så viktig att utvecklaren väljer att använda sig av parallell programmering är personen i fråga antagligen så medveten om prestandan att denne även vill optimera sina samlingsdatatyper för ändamålet. Det är därför viktigt att göra noggranna studier av operationerna på dessa samlingsdatatyper för att kunna välja rätt algoritmer för dessa operationer.

En intressant egenskap dessa samlingsdatatyper kan ha är så kallad konsistensgaranti, denna garanti beskrivs av Herlihy och Shavit (2008). För iteratorer innebär detta att det tillstånd som itereras är ett atomiskt snapshot av kön. Som i sin tur betyder att det tillstånd som itereras är en ögonblicksbild av hur kön såg ut vid någon fast tidpunkt.

Fokus ligger därför på att studera prestandan hos ett flertal kända algoritmer för parallell iterering över samlingsdatatypen kö. Ett par av de iteratorer som testas har testats tidigare men då i en Java miljö (Nikolakopoulos, Gidenstam, Papatriantafilou & Tsigas 2013). Iteratorn hos den parallella kö som implementerats utefter universella metoder för att konstruera parallella dataobjekt från sekventiella (Herlihy 1993; Anderson & Moir 1999) har inte tidigare testats.

Eftersom det är en del ord som saknar en bra svensk översättning inom området, används några engelska ord. Dessa ord finns översatta och med en kort förklaring i bilaga A.

1.2 Problem

På grund av att processorutvecklingen har tagit den väg den har så behöver parallella datastrukturer användas. Det är viktigt för utvecklare att veta hur prestanda och beteende kan skilja sig mellan de algoritmer som finns att tillgå, för att kunna optimera sina system. Kostnaden för att få konsistensgarantier kan spela en stor roll vid valet av algoritm för utvecklare. Det är också intressant för utvecklare att veta hur olika förutsättningar kan påverka algoritmernas prestanda, exempelvis initial storlek och pinning strategi. Initial storlek beskriver hur många element som befinner sig i kön och pinning strategi beskriver vilken kärna respektive tråd skall bindas till.

1.3 Frågeställning

Huvudsyftet med den här studien är därmed att konstruera en handbok för utvecklare. En handbok som kan användas för att göra informerade val kring vilken kö med iterator att använda. För att framställa denna skall följande frågor besvaras:

- Hur stor trade-off är det mellan prestanda och garanti för konsistens?
- Hur förhåller sig prestandan på dessa köer mot varandra?
- Hur stor påverkan på prestandan har olika pinning strategier, initiala storlekar samt trådantal?

1.4 Avgränsningar

Då den här studien har utförts med en viss tidsram har vissa avgränsningar fått göras. Dels så har antalet köer med iterator som testats fått begränsas för att vi författare skulle hinna implementera och testa dem. Även längden på experimenten har fått begränsas för att passa inom tidsramen.

2 Metod

I detta kapitel kommer det att beskrivas hur undersökningen skall utföras. Det beskrivs vilken data som är intressant, hur den samlas in och hur den sedan analyseras.

2.1 Mikrobenchmark

Benchmarking är när prestandan hos ett program eller delar av ett program testas. Det som används i den här studien är så kallade mikrobenchmarks. Detta är när prestandan hos en liten del av programmets kod mäts isolerat. Prestandan mäts alltså inte i förutsättningar som är troliga att finna i en ”verklig” tillämpning. I detta fall är det främst iteratorns prestanda som är intressant att mäta, men även enqueue och dequeue operationernas prestanda.

Programspråket C# använder sig av JIT-kompilering. Det betyder att koden kompileras först när den anropas, vilket kan påverka prestandan. McGachey och Hosking (2006) använde sig av en uppvärmningsomgång innan den första av sina benchmarkkörningar. Det vill säga att innan första benchmarken startar kör den ett varv för att kompilera koden. Under denna körning sparas inte något resultat utan dessa slängs direkt. För att undvika JIT-kompileringens påverkan på prestandan har denna metod tillämpats även här. Genom att göra så här mäts prestandan under så snarlika förhållanden som möjligt under samtliga experiment, för att få så deterministiska resultat som möjligt. Antalet gånger benchmarken kör med samma parameteruppsättning har satts till tio. Totalt blir det 11 körningar, men endast resultaten från 10 körningar används och sparas. Antalet 10 valdes för att vi författare tror att det skall vara tillräckligt många för att få stabila resultat samt att McGachey och Hosking (2006) använde detta antal i sina experiment.

Delar av steady-state performance tekniken framtagen av Georges, Buytaert och Eeckhout (2007) skall användas. Det som tas från denna teknik är att för varje parameteruppsättning körs först en uppvärmningsomgång där resultat inte sparas. Efter detta körs benchmarksen som vanligt och resultat sparas.

Eftersom C# har en inbyggd garbage collector vill vi kontrollera förutsättningarna för denna under våra mätningar, då den kan påverka mätningarna. Detta undviks genom att köra den manuellt före varje benchmark. Även detta gör att förhållandena inför varje experiment blir så snarlika som möjligt.

2.1.1 Parametrar

Benchmarken som kommer att användas i experimenten kommer ta ett flertal olika parametrar. Dessa går att utläsa från Tabell 2.1.

Parameter	Beskrivning
Initial storlek	Hur många element kön fylls med innan experimenten startar.
Antal trådar	Antalet trådar som skall användas under experimenten, inklusive iteratortråd.
Kö	Vilken köimplementation som skall användas vid experimentet.
Pinning strategi	Vilken pinning strategi som skall användas vid experimenten.
Tid	Hur lång tid varje benchmark kör i millisekunder.
Upprepningar	Hur många benchmarks som skall köras för varje parameteruppsättning, exklusive uppvärmningskörningen.

Tabell 2.1 – Parametrar för benchmarkprogrammet

Initial storlek är en intressant parameter eftersom olika värden på denna parameter ger olika stor chans för kön att bli tom under experimenten. Det är även intressant att undersöka om storleken på kön påverkar de olika operationernas prestanda. Antal trådar blir intressant för att testa köns olika operationer under olika stora belastningar. Olika köer kommer testas för att jämföra prestandan och beteende hos de olika implementationerna. Tiden som benchmarkprogrammet kommer att köra bör vara tillräckligt lång för att utjämna olika ojämnheter i empirin. Samtidigt skall tiden vara tillräckligt kort för att experimenten ska vara utförbara inom studiens tidsram. Upprepningar följer samma mönster som tiden, men då att det skall vara tillräckligt många upprepningar för att säkerställa ett reliabelt resultat. Motivering för varför olika pinning strategier kommer testas följer i avsnittet nedan.

2.1.2 Pinning

Vid arbete med så många trådar som faktiskt kommer att användas i experimentet kan det tänkas påverka prestandan vilken kärna i processorn som varje tråd kör på. Istället för att låta Windows bestämma vilken kärna varje tråd skall köras på kommer så kallad pinning användas. För att undersöka om prestandan påverkas kommer det i experimenten användas ett flertal olika pinning strategier. Detta innebär att varje tråd har låsts till en specifik logisk kärna, och kommer därmed att köras på denna kärna. Detta ger vid max antal trådar (24st) en tråd per logisk kärna i processorn.

2.2 Insamling av data

Det finns ett flertal olika värden som är intressanta. Vilka dessa är och en beskrivning av dem följer nedan. Dessa sparas för varje benchmark i benchmarkprogrammet, det vill säga 10 gånger för varje parameteruppsättning.

- Lyckade och misslyckade enqueue samt dequeue operationer.
  Genom antalet misslyckade dequeue operationer är det lätt att se hur många gånger kön är tom. Summan av antalet lyckade och misslyckade enqueue och dequeue operationer ger ett värde på hur många operationer arbetstrådarna har hunnit med under körningstiden. Denna data mäts för att få ett mått på hur effektiva enqueue och dequeue operationerna är hos de olika köerna. Största anledningen att mäta denna prestanda är för att få en helhetsbild över hur bra kön presterar. Antalet misslyckade dequeue operationer mäts för att få ett mått på hur ofta kön blir tom. Detta är intressant att mäta dels så att man kan justera vilka initiala storlekar som skall testas. Men även för att se hur mycket antalet element i kön varierar.

- Faktisk tid som benchmarken kör.
  Benchmarken mäter den faktiska tiden tills alla trådar har kört klart. Denna tid kan skilja sig mot den specificerade tiden, då samtliga trådar får avsluta sina pågående operationer när den specificerade tiden passerat. Denna tid mäts för att upptäcka ifall någon operation hos någon av de olika köerna vid något tillfälle fastnar en längre tid. Det kan vara intressant att se om någon operation i värsta fall kan ta väldigt lång tid.

- Antalet iterationer som iterator tråden hinner genomföra.
  Antalet iterationer är trivialt för att uppskatta iteratorns prestanda. Detta antal mäts för att få ett mått på prestandan hos iteratorn. Detta mått på prestanda varierar mycket beroende på antalet element som befinner sig i kön vid iterationen. Med detta mått och antalet element som itereras över i varje iteration kan man sedan beräkna antalet itererade element per sekund om så önskas.

- Antal luckor som påträffas i kön av iterationstråden.
  Dessa luckor visar främst om något är fel i implementationen av de olika köerna. Utöver detta går det även att fastslå att elementen itereras i rätt ordning. Detta antal mäts för att se om någon av de testade köerna inte itererar elementen i rätt ordning.

- Antalet element som itereras över i varje iteration.
  Med denna data beräknas sedan hur många element som itereras som minst och som högst, samt ett medelvärde. Med hjälp av detta medelvärde och antalet utförda iterationer går det att beräkna antalet itererade element per sekund. Det sparas även antalet element som itererades över i sista iterationen. Detta antal mäts för att få en bild över hur många element som befinner sig i kön under experimentens gång. Med hjälp av detta går det att se om prestandan hos de olika operationerna är beroende på antalet element som befinner sig i kön.

För att säkerställa resultatens reliabilitet kommer samtliga experiment att köras två gånger. Först kommer en första insamling för att få testdata att analysera att utföras. Därefter kommer en andra insamling utföras för att konfirmera att datan från första körningen stämmer. Detta ihop med att för varje parameteruppsättning körs 10 mikrobenchmarks anser vi författare leder till att resultaten från experimenten skall vara reliabla.

2.3 Analys av data

Datan som produceras av experimenten kommer att analyseras från ett flertal olika vinklar. Det kommer analyseras hur olika pinning strategier påverkar både arbetstrådarna och iteratortråden vid olika trådantal. Det undersöks även hur stor påverkan det får att ha varierande antal element i kön initialt. Även hur mycket iteratorn påverkas av antalet element i kön som denna måste iterera kommer analyseras. Till sist kommer de olika köernas prestanda analyseras och jämföras mot varandra. Datan visualiseras och presenteras i form av grafer. I dessa grafer presenteras genomsnittliga värden från de upprepningar som gjorts inom varje parameteruppsättning. Det kommer skapas tre olika grafer. Dessa grafer visualiserar antal iterationer per sekund, genomsnittligt antal enqueue och dequeue operationer per sekund samt genomsnittligt antal element per iteration.

För att mäta enqueue och dequeue operationernas prestanda används måttet antalet utförda operationer per sekund. För dessa operationer anser vi författare att antalet utförda operationer är det intuitiva valet, samt att det saknas vettiga alternativ. Iteratorns prestanda har vi författare

valt att mäta med antalet utförda iterationer per sekund. Det alternativ som resonerades med var att istället mäta det med antal itererade element per sekund. Det vi författare ville att detta mått skall representera är mestadels tiden det tar att hämta iteratorn. Vilket vi tycker representeras bäst av antalet utförda iterationer per sekund.

3 Teori

Detta kapitel beskriver parallella datastrukturer, de olika algoritmerna som har använts i experimenten, olika pinning strategier samt vad benchmarking är. Det tas även upp vilka relaterade arbeten som finns och hur dessa har influerat denna studie. Vilken hårdvara som använts i experimenten beskrivs även kort i detta kapitel.

3.1 Parallella datastrukturer

När fler och fler program börjar använda sig av multitrådning blir det vanligare att även behöva dela data mellan ett flertal trådar. Detta kan då enklast göras med hjälp av parallella datastrukturer. En parallell datastruktur är en datastruktur som kan både läsas från och skrivas till från flera trådar samtidigt.

Herlihy och Shavit (2008) beskriver i sin bok ett par olika garantier som operationer hos parallella datastrukturer kan ge. Framstegsgarantier berör hur operationen kan garantera att framsteg nås och finns i tre nivåer. Den lägsta nivån av denna garanti kallas blockerande (eng. blocking). För denna garantinivå kan en tråds oväntade fördröjning orsaka att andra trådar inte kan framskrida. Operationer som använder sig av lås ligger i denna garantinivå. Nästa nivå av garantin kallas låsfri (eng. lock-free). Det är svårt att precisera när garantinivån introducerades men har beskrivits av Barnes (1993) och Herlihy (1993). En operation har denna garantinivå om den kan garantera att minst en av de trådar som försöker utföra operationen kommer att lyckas inom ett ändligt antal steg. Den högsta garantinivån kallas för väntefri (eng. wait-free) och föreslogs först av Herlihy (1991). Om operationen även kan garantera att samtliga tävlande trådar kommer att lyckas inom ett ändligt antal steg anses operationen vara väntefri.

Herlihy och Shavit (2008) beskriver även en garanti för konsistensen hos datastrukturens semantik, denna garanti kallas för linjäriserbarhet och föreslogs först av Herlihy och Wing (1990). En operation anses vara linjäriserbar om operationen ser ut att utföras ögonblickligt vid någon tidpunkt under dess körning. Om en iterator är linjäriserbar lämnar denna ett atomiskt snapshot av datastrukturen. Ett atomiskt snapshot kan beskrivas som en bild av hur datastrukturen såg ut vid någon fast tidpunkt. De flesta av de iteratorer som använts i testerna lämnar garantier för att det tillstånd som returneras är ett atomiskt snapshot. Den enda iterator som inte gör det är Scan and Return iteratorn.

3.2 Algoritmer

Det finns många olika datastrukturer som går att testa prestanda och beteende hos. Det finns de som ger utvecklaren fri tillgång till elementen i samlingen, exempelvis en lista, eller en hashmap. Det finns även datastrukturer där utvecklaren endast kan komma åt vissa element, typiskt första och/eller sista. Exempel på dessa datastrukturer är kö, stack och heap. Vi författare valde att testa datastrukturen kö för att denna studie har en stark koppling till Nikolakopoulos et al. (2013) arbete. I deras artikel föreslog de ett flertal iteratorer till datastrukturen kö, som de även testade i en Java miljö. Då vår studie är något av en fortsättning på deras föll det naturligt att testa datastrukturen kö. Detta ger även möjligheten att delvis jämföra resultaten från vår studie med deras.

Det finns ett flertal olika ansatser till parallella köer, dessa presenteras av Cederman, Gidenstam, Ha, Sundell, Papatriantafilou och Tsigas (2013) i deras samlingsverk. Det finns köer som bygger på en statisk vektor (Lampart 1983; Giacomoni, Moseley & Vachharajani 2007; Herman & Damian-Iordache 1997). Det finns även köer som bygger på en cyklisk vektor (Gong & Wing 1990, Chann, Huang & Chen 2000). Även en kö som använder sig av en kombination av vektorer och en länkad lista finns (Gidenstam, Sundell & Tsigas 2010). Även

köer som bygger på en länkad lista finns (Valois 1994; Valois 1996; Prakash, Lee & Johnson 1994; Michael & Scott 1996; Moir, Nussbaum, Shalev & Shavit 2005; Hoffman, Shalev & Shavit 2007). Det finns dock väldigt få parallella köer med iteratorer. Den enda av ovan nämna köer som en iterator har föreslagits till är Michael och Scott kön (Michael & Scott 1996).

Eftersom Michael och Scott kön (Michael & Scott 1996) som sagt är den enda av ovan nämnda köer som har iteratorer föreslagna, har denna valts ut för att testas. Till denna kö har två stycken iteratorer föreslagna och testade i en Java miljö av Nikolakopoulos et al. (2013) valts ut för att testas: Scan and Return och Double Collect. Detta blir då två stycken köer med iterator, en Michael och Scott kö med Scan and Return iterator och en Michael och Scott kö med Double Collect iterator. Nikolakopoulos et al. (2013) föreslog i sin studie fyra stycken iteratorer till Michael och Scott kön. Eftersom tid inte fanns att implementera och testa samtliga fyra iteratorer valdes två ut att använda. Då en del av frågeställningen var att undersöka kostnaden för konsistensgaranti valdes Scan and Return och Double Collect. Dessa två iteratorer är väldigt lika varandra bortsett från att Double Collect lämnar konsistensgaranti medan Scan and Return inte gör det. Detta gör då att de här iteratorerna lämpar sig bra till att mäta kostnaden för konsistensgarantin. Utöver dessa två köer med iterator har två till köer med iterator testats. Den parallella kö med iterator som finns i .NET biblioteket, ConcurrentQueue (Microsoft u.å. a), är en av dessa. Den valdes av ett par olika anledningar. Dels kan denna kö användas som en bedömningsgrund vid jämförelse med de övriga testade köerna. Denna kö lämpar sig bra som bedömningsgrund då denna kö alltid finns tillgänglig för utvecklare i en .NET miljö. Det är även intressant att se hur väl Microsoft har optimerat prestandan i denna implementation av kö med iterator. Den sista kön med iterator som testats har konstruerats från universella metoder för att konstruera parallella dataobjekt från sekventiella (Herlihy 1993; Anderson & Moir 1999). Denna implementation av kö med iterator använder sig av den immutable kö som finns tillgänglig i .NET biblioteket version 4.5 och senare, ImmutableQueue (Microsoft u.å. d), för att representera en version av kön. En Immutable kö är en kö som inte kan ändras på efter initiering. Modifierande operationer returnerar istället en ny instans av kön med de nya ändringarna. Denna kö valdes att vara med i testerna för att det är en annan ansats till parallell kö med iterator än de tidigare tre. Då vi författare ville testa så många olika ansatser som möjligt blev det ett enkelt val att inkludera denna kö i testerna.

I de olika köernas implementationer används den atomiska operationen Compare And Swap (CAS). Att en operation är atomisk innebär att hela operationen utförs ögonblickligt. Denna operation tar tre argument; destination, komparand och utbyte. Det operationen gör är att den jämför destinationen med komparanden. Om dessa två värden stämmer överens skrivs utbytet till destinationen.

3.2.1 Michael och Scott

En av köerna som använts är framtagen av Michael och Scott (1996). Denna kö är baserad på en länkad lista och innehåller endast enqueue och dequeue operationer. Mindre modifikationer har gjorts på kön då C# har inbyggd garbage collector. På grund av denna garbage collector behövs inte minnet släppas manuellt som det behövdes i original implementationen. Michael (2004) har tagit fram en version av kön som är anpassad till miljöer med garbage collector.

Kön är internt representerad som en länkad lista, där det finns en pekare till huvud och svans. Huvudpekaren pekar alltid på en så kallad dummynod. Det är alltså en nod som inte har något värde och vars next-pekare pekar på första elementet som innehåller ett värde. Denna dummynod finns för att kunna representera en tom kö på samma sätt som en icke-tom kö. Svanspekaren pekar alltid på sista elementet i kön eller näst sista elementet under en enqueue

operation. Varje enskild nod innehåller ett värde och en next-pekare som pekar på nästa element i kön.

Orginalet av kön innehåller dock ingen iterator, men sådana har senare föreslagits. Joe Duffy (2009) föreslog en iterator i sin C# implementation av Michael och Scott kön. Nikolakopoulos et al. (2013) föreslog två iteratorer de kallar Scan and Return och Double Collect. Joe Duffys (2009) iterator och Scan and Return iteratorn är lika varandra. De två iteratorer som är implementerade till testerna för denna kö är Scan and Return och Double Collect. Skillnaderna mellan dessa iteratorer är huruvida de kan garantera ett atomiskt snapshot av kön eller inte.

Class 1 Michael och Scott kö

```
Struct Node
{
   Value : DataType
   Next : *Node
}

//Shared variables
Head, Tail : *Node

 1.  Enqueue(item)
 2.  {
 3.     node ← new Node()
 4.     node.Value ← item
 5.     node.Next ← null
 6.     while(true)
 7.     {
 8.         tail ← Tail
 9.         next ← tail.Next
10.         if (Tail ≠ tail)
11.         {
12.             continue
13.         }
14.         if (next ≠ null)
15.         {
16.             CAS(&Tail, tail, next)
17.             continue
18.         }
19.         if (CAS(&tail.Next, null, node))
20.         {
21.             break
22.         }
23.     }
24.     CAS(&Tail, tail, node)
25. }

26. Dequeue()
27. {
28.    while(true)
29.    {
30.        head ← Head
31.        tail ← Tail
32.        next ← head.Next
33.        if (Head ≠ head)
34.        {
35.            continue
36.        }
37.        if (next = null)
38.        {
39.            return failure
40.        }
41.        if (head = tail)
42.        {
43.            CAS(&Tail, tail, next)
44.            continue
45.        }
46.        data ← next.Value
47.        if (CAS(&Head, head, next))
48.        {
49.            break
50.        }
51.    }
52.    return data
53. }
```

Scan and Return

Scan and return är en iterator föreslagen av Nikolakopoulos et al. (2013). Den hämtar först en pekare till huvudet på kön och sedan en pekare till svansen av kön. Den kontrollerar inte vid något tillfälle huruvida kön har ändrats parallellt eller inte. Med andra ord så kan en annan tråd modifiera kön mellan det att huvudet hämtas och det att svansen hämtas utan att detta upptäcks. Denna iterator kan alltså inte lämna någon garanti för att det tillstånd som returneras är ett atomiskt snapshot av kön. Bevisning av avsaknaden av denna garanti görs av Nikolakopoulos et al. (2013). Modifikationer på kön efter att iteratorn hämtats återspeglas inte i iterationen.

Den iterator Joe Duffy (2009) föreslog i sin implementation av Michael och Scott kön är likadan som Scan and Return iteratorn med en skillnad. Joe Duffys iterator kontrollerar om svanspekaren halkat efter och hjälper den i så fall att komma ikapp, vilket Scan and Return inte gör. Joe Duffy (2009) hävdade att denna iterator gav ett atomiskt snapshot av kön. Detta stämmer dock inte då skillnaderna mellan denna iterator och Scan and Return iteratorn inte påverkar konsistensen. Eftersom Nikolakopoulos et al. (2013) har bevisat att Scan and Return inte lämnar garanti för atomiskt snapshot kan vi dra slutsatsen att inte heller Joe Duffys iterator gör det.

Algoritm 1 Scan and Return

```
1.  GetIterator()
2.  {
3.     StateToReturn.Initialize()
4.     StateToReturn.head ← Head
5.     StateToReturn.tail ← Tail
6.
7.     return StateToReturn
8.  }
```

Double Collect

Double Collect är en annan iterator framtagen av Nikolakopoulos et al. (2013), som ger mer garantier. Skillnaderna från Scan and Return iteratorn är att den efter att ha hämtat pekare till både huvud och svans kontrollerar att pekaren till huvudet fortfarande är samma. Den kontrollerar även att svansen faktiskt är det sista elementet i kön. Dessa två kontroller ger en garanti att kön vid tillfället då pekaren till svansen hämtades såg ut exakt som det tillstånd som returneras. Denna iterator kan alltså lämna en garanti för att det tillstånd som returneras är ett atomiskt snapshot av kön, med synkroniseringspunkt där pekaren till svansen hämtas (rad 7 i Algoritm 2). Bevisning av denna garanti görs av Nikolakopoulos et al. (2013). Modifikationer på kön efter att iteratorn hämtats återspeglas inte i iterationen.

Algoritm 2 Double Collect

```
1.  GetIterator()
2.  {
3.     while (true)
4.     {
5.         StateToReturn.Initialize()
6.         StateToReturn.head ← Head
7.         StateToReturn.tail ← Tail
8.         Next ← StateToReturn.tail.next
9.
10.        if (Next ≠ null)
11.        {
12.            CAS (Tail, StateToReturn.tail, Next)
13.            continue
14.        }
15.
16.        If (StateToreturn.head == Head)
17.            return StateToReturn
18.    }
19. }
```

3.2.2 .NET bibliotekets parallella kö

En adapter klass har använts för att kunna testa den parallella kö med iterator som finns i .NET biblioteket (Microsoft u.å. a), ConcurrentQueue, i benchmarkprogrammet. En adapter klass är

en klass som används som en brygga för att få två olika klasser att arbeta ihop utan att modifiera någonderas källkod. Det klassen gör internt är att ha en ConcurrentQueue som den utför samtliga operationer mot, samtidigt som den förlänger det interface som benchmarkprogrammet arbetar mot.

Den iterator som denna kö implementerar garanterar att ett atomiskt snapshot av kön returneras enligt dokumentationen (Microsoft u.å. b). Som en konsekvens av detta kommer inte heller modifikationer på kön efter att iteratorn hämtats återspeglas i iterationen.

3.2.3 Kön som bygger på Immutable

En immutable samlingsdatatyp är en samlingsdatatyp som inte kan ändras på efter initiering, istället skapar de modifierande operationerna ett nytt objekt som sedan returneras. I .NET 4.5 och senare finns det tillgång till en mängd olika immutable samlingsdatatyper (Microsoft u.å. c). Den kö som finns tillgänglig (Microsoft u.å. d) har använts för att representera en version av kön vid tillämpning av universella metoder att konstruera parallella dataobjekt från sekventiella (Herlihy 1993; Anderson & Moir 1999). Även denna kö har en iterator. Vid denna tillämpning har nya operationer för kön implementerats. De operationer som skrivits är enqueue, dequeue samt iterator. Dessa operationer arbetar mot Immutable kön (Microsoft u.å. d) och underhåller köns nuvarande tillstånd.

Algoritm 3 Enqueue för kö baserad på ImmutableQueue

```
1.  Enqueue(item)
2.  {
3.     while(true)
4.     {
5.         OldState ← CurrentState
6.         NewState ← OldState.Enqueue(item)
7.         if (CAS(CurrentState, OldState, NewState))
8.             return true
9.     }
10. }
```

Algoritm 4 Dequeue för kö baserad på ImmutableQueue

```
1.  Dequeue()
2.  {
3.     while(true)
4.     {
5.         OldState ← CurrentState
6.         if (OldState.isEmpty)
7.             return failure
8.         NewState ← OldState.Dequeue(item)
9.         if (CAS(CurrentState, OldState, NewState))
10.            return item
11.    }
12. }
```

Algoritm 5 Iterator för kö baserad på ImmutableQueue

```
1. GetEnumerator()
2. {
3.    EnumeratedState ← CurrentState
4.
5.    foreach (Item in EnumeratedState)
6.    {
7.        yield return Item
8.    }
9. }
```

Samtliga av dessa operationer är linjäriserbara. Enqueue och dequeue operationerna uppnår detta genom sina CAS operationer. Då operationerna inte modifierar CurrentState annat än med CAS som är en atomisk operation uppnås linjäriserbarhet. Eftersom iteratorn läser köns tillstånd vid en fast tidpunkt och detta lästa tillstånd är garanterat att inte förändras, är även denna linjäriserbar.

Det finns ont om dokumentation från Microsoft om hur deras immutable implementationer fungerar internt. Vi har undersökt Eric Lipperts blogg (Lippert 2007) där det beskrivs just en immutable kö med iterator, dock utan någon direkt koppling till den implementation som finns i .NET. Även den .dll fil som innehåller .NET's immutable kö har packats upp med verktyget IL DASM för att kunna jämföra denna med Lipperts kö. Eric Lippert är före detta anställd hos Microsoft som principal developer i C# teamet.

Eric Lipperts kö (Lippert 2007) är internt uppbyggd av två stackar. Elementen läggs till i den så kallade backward stacken vid en enqueue operation. De tas bort från den så kallade forward stacken vid en dequeue operation. Om forward stacken är tom vid en dequeue operation flyttas samtliga element från backward stacken till forward stacken. Elementen flyttas ett och ett vilket leder till att de hamnar i omvänd ordning i forward stacken. Detta leder till att varje element läggs till och tas bort från de båda stackarna endast en gång. Iteratorn i Lipperts kö fungerar genom att först iterera forward stacken, följt av att iterera backward stacken baklänges. Detta kan tänkas kräva en del prestanda vid stora köer eftersom det är kostsamt att vända på en stack och iterera den baklänges.

Vid uppackning av .NET's immutable kö och inspektion av dess intermediate language syns det en hel del likheter mot Lipperts kö (Lippert 2007). .NET's immutable kö har internt tre stycken stackar. Den har både forward och backward likt Lipperts kö, den har även en backwardReversed stack. Eftersom det inte finns någon dokumentation kring hur denna kö arbetar internt är följande resonemang endast spekulationer från författarnas sida. Vi författare antar att den kö som finns i .NET fungerar på samma sätt som Lipperts kö, med den tillagda backwardReversed stacken. Denna stack tror vi författare används för att spara den vända backward stacken för att kunna återanvända denna, mestadels då vid iteration. Det kan dock tänkas att den sparade stacken inte kommer till stor använding under experimenten, eftersom det i experimenten utförs många mer enqueue och dequeue operationer än iterationer. Därför går det mycket väl att tänka sig att den vända stacken som sparas vid en iteration inte längre stämmer överens med backward stacken vid nästa iteration.

3.3 Pinning

CPU-pinning är när programmeraren bestämmer vilken processoraffinitet en process eller tråd skall ha. Det är en teknik som låter utvecklare binda en process eller en tråd till en eller flera

specifika kärnor i processorn. Det leder till att tråden, eller processen, garanterat kommer att köras på de specificerade kärnorna. Operativsystemet kan alltså inte flytta tråden till en kärna som den anser vara bättre lämpad att köra den. Nyttan i att göra detta är framför allt att utnyttja det faktum att information som tråden behöver kan vara kvar i processorns cache minne. Men även för att slippa att trådarna ska byta kärna, då detta påverkar prestandan.

3.4 Benchmarking

I det här avsnittet kommer det diskuteras hur mikrobenchmarksen är utförda och även motivera varför de är gjorda på detta sätt. Det kommer diskuteras vilken data som är intressant att samla in, samt under vilka förutsättningar dessa uppmätningar sker. Här kommer även ges en övergriplig förklaring till vissa uttryck som används i dessa diskussioner.

En benchmark är en körning med en viss parameteruppsättning. Denna körs ett flertal gånger, som är specificerat av användaren. Benchmarkprogrammet är alla körningar med en specificerad parameteruppsättning. I testerna är dessa 10 stycken, exklusive en uppvärmningsomgång.

3.4.1 Benchmarkprogrammet

Vid uppstart av programmet finns det ett flertal inställningar att göra. Det går här att välja vilken kö som skall testas och hur många arbetstrådar som ska arbeta mot kön. Det går även välja hur länge varje iteration av benchmarken skall köra, samt hur många element som skall finnas i kön vid början av benchmarken.

En körning av programmet består då av att så många benchmarks som önskas körs. Utöver detta antal körs en extra benchmark i början för att de metoder som används skall kompileras av C#'s JIT, resultaten för denna körning sparas inte. Före varje benchmark initieras kön om på nytt och fylls med så många element som specificerats. Efter detta körs garbage collectorn manuellt, detta för att få lika förutsättningar som möjligt inför varje körning så att resultaten blir så deterministiska som möjligt. Varje körning av benchmarken, eller iteration, består av att så många arbetstrådar som specificerats samt en iterationstråd skapas. Sedan signaleras samtliga trådar att starta samtidigt. Benchmarken kör sedan så länge som specificerats. När denna tid har löpt ut signaleras samtliga trådar att avsluta sitt arbete. Trådarna gör då färdigt sin nuvarande operation och efter det terminerar de, de får alltså tid att avsluta det arbete de håller på med.

3.4.2 Arbetstråd

En arbetstråd är en tråd som utför arbete på kön. Det arbete den utför är att slumpmässigt göra enqueue eller dequeue, med lika stor sannolikhet på den aktuella kön. När dessa trådar skapas tilldelas de ett unikt ID.

3.4.3 Mätning

Benchmarken mäter bland annat den faktiska tiden som benchmarken är igång. Denna tid kan komma att variera även när körningen har ett utsatt tidsintervall, då trådarna i benchmarken får köra färdigt istället för att avsluta mitt i en operation. Utöver tiden mäts även antalet enqueue och dequeue operationer, både lyckade och misslyckade. Även antalet iterationer som hanns med samt hur många element som itererades i varje iteration mäts. För varje parameteruppsättning har testerna upprepats 10 gånger, för att fånga ojämnheter i empirin. Dessa ojämnheter kan framkallas av bland annat slumpen och Windows olika bakgrundsprocesser.

3.4.4 Förutsättningar

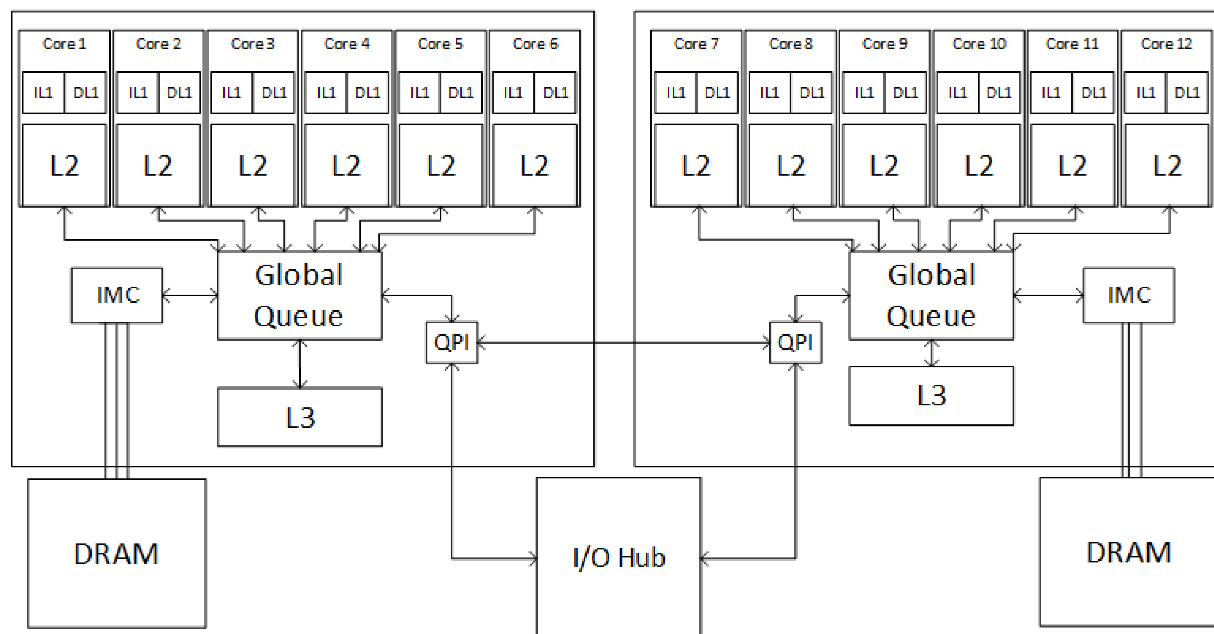
Experimenten är utförda under ett flertal olika förutsättningar. Vilka olika parametrar som använts går att se i Tabell 2.1. Alla trådar i experimenten har samma prioritet. Före varje körning av benchmarken kommer kön att initieras om och fyllas med data upp till initial storlek. Därefter kommer en full garbage collection att köras.

3.5 Hårdvara

Samtliga tester i denna studie har körts på en och samma dator. Datorn är en Dell precision T7500 som är utrustad med två stycken sexkärniga Intel® Xeon™ E5660 CPU @ 2.8 GHz processorer med hypertrådning, detta ger totalt 24 stycken logiska kärnor. Datorn är utrustad med 12Gb fysiskt minne och kör operativsystemet Microsoft Windows 7 Ultimate Service pack 1 version 6.1.7601 . Benchmarkprogrammet som använts under testerna är utvecklat i Microsoft Visual Studio Professional 2013 version 12.0.21005.1 REL i Microsoft .NET Framework version 4.5.50938.

3.5.1 Processorns minneshierarki

Eftersom experimenten har testat påverkan av olika pinning strategier är det intressant att undersöka hur minneshierarkin för den aktuella processorn ser ut. Thomadakis (2011) har studerat minneshierarkin hos nehalem processorer. Det framgår i hans studie att dessa processorer har delat L3 cache för hela processorn. Varje fysisk kärna har sitt eget L2 och L1 cache. Detta betyder alltså att det minne som olika kärnor inom samma processor vill dela synkroniseras i L3 cachen.



Figur 3.1 – Minneshierarki för nehalem processorer efter Thomadakis (2011, s.31)

Figur 3.1 visar minneshierarkin för nehalem processorer. Thomadakis (2011, s.31) visar en figur för en processor med 4 kärnor, men hierarkin ska vara densamma eftersom det fortfarande rör sig om nehalem arkitekturen.

3.6 Relaterade arbeten

En liknande studie har utförts av Nikolakopoulos et al. (2013). Denna innefattade de teoretiska aspekterna av iteration med avseende på synkronisering och konsistens, den föreslår också fyra

stycken iteratoralgoritmer för köer. Två av dessa iteratorer kallar de för Scan and Return och Double Collect. De övriga två är namnlösa och bygger på att iteratorn får hjälp. Den ena får hjälp av enqueue och den andra av enqueue och dequeue. I deras studie utfördes samtliga experiment i en Java miljö. En Java miljö är relativt lik en C# miljö, då båda två använder en garbage collector och JIT-kompilering.

Georges, Buytaert och Eeckhout (2007) diskuterade i sin artikel de problem som uppstår när man vill uppnå determinism i programspråk som använder sig av garbage collector och JIT-kompilering. Även de komplikationer som en schemaläggare för trådar tillför tas upp. Deras lösning på problemen är att mäta steady-state performance. Denna teknik tar hänsyn till JIT-kompileringen genom att man låter benchmarken köra en gång först där mätresultat inte samlas in. Sedan körs det antal benchmarks som önskas där mätresultat samlas in och analyseras. Denna första körning sätter då igång JIT-kompileringen så att samtlig kod finns färdigkompilerad när nästföljande benchmarks körs. Varje benchmark borde enligt tekniken vara tillräckligt lång för att jämna ut eventuella ojämnheter, exempelvis garbage collectorn. Schemaläggarens och andra utomstående parters påverkan berörs dock inte av denna teknik utan måste tas i hänsyn på annat sätt.

Vi författare har i studien valt att testa två av de iteratorer som Nikolakopoulos et al. (2013) föreslog. Dessa är Scan and Return samt Double Collect. Vi har även valt att tillämpa delar av steady-state performance tekniken. Här körs en benchmark först utan att samla in data för att JIT-kompilera samtlig kod, en så kallad uppvärmningsomgång. Följt av att 10 stycken till benchmarks körs som samlar in data. Garbage collectorns påverkan jämnar vi ut genom att köra den före varje benchmark. På detta vis används lika mycket minne vid varje benchmarks start. Schemaläggarens påverkan hanteras genom de olika pinning strategierna som testats.

4 Resultat

Detta kapitel berör resultaten av experimenten samt hur benchmarkprogrammet och de olika köerna med iteratorer har implementerats. Det tas upp hur experimenten utförts och vilka förutsättningar som funnits. Under implementationsdelen förklaras mer utförligt hur benchmarkprogrammet fungerar, hur trådarna har bundits till en specifik logisk kärna samt hur köerna översatts från pseudokod till C# kod.

4.1 Val av parametervärde

Tabellen nedan visar vilka värden de olika parametrarna för benchmarken har fått under experimenten. Fyra av totalt sex parametrar har haft varierande värden.

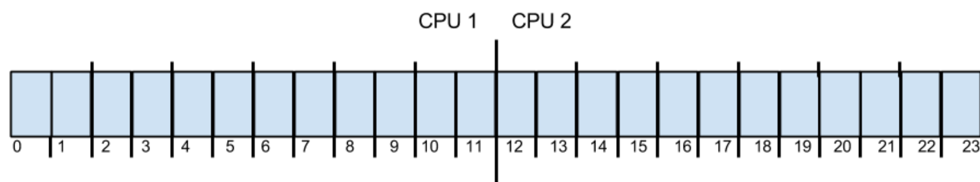
Parameter	Beskrivning	Värden
Initial storlek	Hur många element kön fylls med innan experimenten startar.	0, 5 000, 10 000
Antal trådar	Antalet trådar som skall användas under experimenten, inklusive iteratortråd.	2, 4, 8, 10, 12, 14, 16, 20, 22, 24
Kö	Vilken köimplementation som skall användas vid experimentet.	Scan and Return, Double Collect, Framework, kön baserad på immutable
Pinning strategi	Vilken pinning strategi som skall användas vid experimenten.	None, Alternate, Fill
Tid	Hur lång tid varje benchmark kör i millisekunder.	20 000
Upprepningar	Hur många benchmarks som skall köras för varje parameteruppsättning, exklusive uppvärmningskörningen.	10

Tabell 4.1 – Parametrar för benchmarkprogrammet med antagna värden

Vid en testomgång av benchmarkprogrammet användes de initiala storlekarna 0, 2 000, 5 000. Det visade sig här att kön fortfarande blev tom relativt ofta vid 5 000 så de initiala storlekarna fick höjas till 0, 5 000 samt 10 000. Trådantal har stegats med 4, men med mindre steg runt 12 och 24 trådar. De höga trådantalen ger en bild av värsta fall för kön. Antalet upprepningar sattes till 10 då McGahey och Hosking (2006) använde sig av detta. Tiden sattes till 20 sekunder då en uppvägning gjordes för hur lång tid experimenten får ta för att studien skall hinna färdigställas. Motivering av varför pinning strategierna none, alternate och fill har använts följer nedan. Motivering av val av de köer som använts finns i *3.2 Algoritmer*.

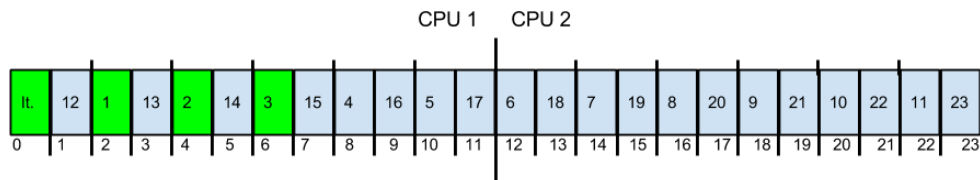
4.1.1 Pinning

För att använda pinning tekniken i C# används ProcessorAffinity propertyn på ProcessThread klassen (Microsoft u.å. f). Med hjälp av denna property går det att specificera vilken eller vilka kärnor en specifik tråd skall köras på. I experimenten har processorernas kärnor numrerats enligt Figur 4.1. Plats noll ockuperas vid strategierna alternate och fill alltid av iteratortråden.

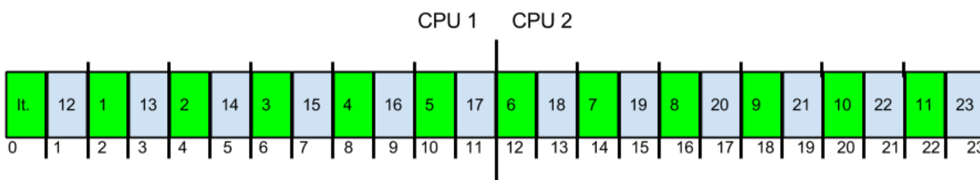


Figur 4.1 – Numrering av processorernas kärnor

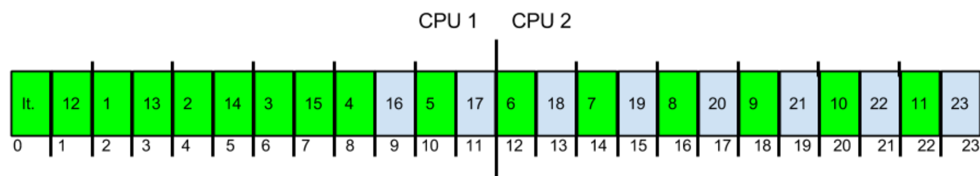
Den första strategin som har testats är ”none”, alltså att låta Windows själv sköta trådhanteringen. Ingen manuell låsning av trådarna till en specifik kärna har använts här. Denna strategi testas eftersom det är intressant att se hur bra Windows är på att fördela arbetet. Strategin uppfyller även funktionen som bedömningsgrund vid jämförelse med övriga strategier. Den andra strategin som testats är ”alternate”. Detta är när varje fysisk kärna får varsin tråd till dess att trådantalet är högre än antalet fysiska kärnor, i detta fall 12. Därefter kommer hypertrådning att användas och varje fysisk kärna får 2 trådar, det vill säga varje logisk kärna får varsin tråd. Se Figur 4.2, Figur 4.3 & Figur 4.4. Den här strategin valdes för att se hur bra prestanda som kan uppnås om man låter bli att använda hypertrådning tills varje fysisk kärna fått en tråd. Den tredje strategin som testats är ”fill” och går ut på att den första processorn fylls innan den andra kommer att användas. Med andra ord kommer hypertrådning att användas direkt från 2 trådar och uppåt. När trådantalet överstiger antalet logiska kärnor i första processorn, i detta fall 12, kommer den andra processorn att börja användas. Se Figur 4.5, Figur 4.6 & Figur 4.7. Den här strategin testas för att se hur bra prestanda som kan uppnås genom att använda endast en processor tills denna är full.



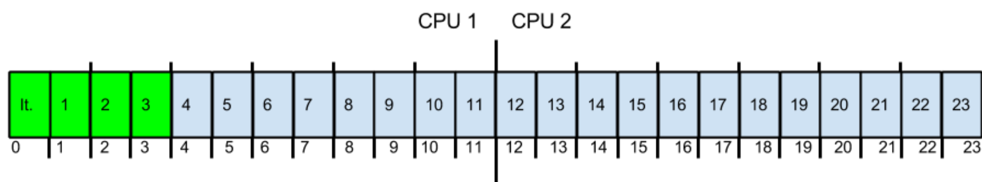
Figur 4.2 – Alternate pinning, 4 trådar



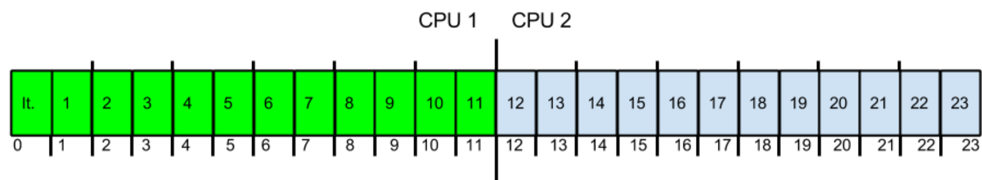
Figur 4.3 – Alternate pinning, 12 trådar



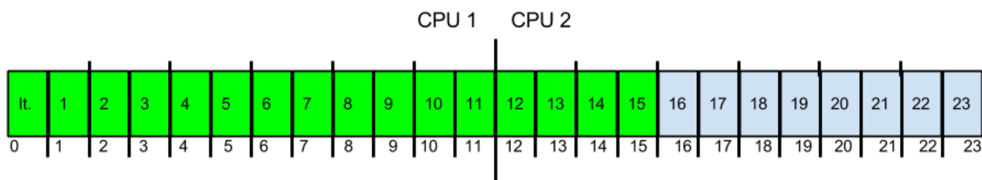
Figur 4.4 – Alternate pinning, 16 trådar



Figur 4.5 – Fill pinning, 4 trådar



Figur 4.6 – Fill pinning, 12 trådar



Figur 4.7 – Fill pinning, 16 trådar

4.2 Implementation

Nedan tas det upp hur det benchmarkprogram som testerna har utförts i har implementerats. Även hur respektive kö och dess iterator har implementerats i förhållande till sin pseudo kod förklaras. All kod som producerats finns bifogad i bilaga B.

4.2.1 Benchmark

Benchmarkprogrammet är implementerat i C# som en konsollapplikation. Den tar in ett flertal parametrar som sätter förutsättningarna för testerna. Även ett hjälpkommando finns att tillgå om så skulle behövas. Varje körning av programmet kör tester för en och endast en parameteruppsättning. Vill det köras tester med flera parameteruppsättningar behöver programmet startas upp flera gånger. Detta har gjorts med ett enkelt script där alla olika kombinationer av parametrar körs.

Ett gränssnitt, IParallelQueue, har implementerats som benchmarken arbetar mot vid testkörningarna. Gränssnittet innehåller tre metoder; Enqueue, Dequeue och CreateNew. Den sistnämnda används när benchmarken vill initiera om kön. IParallelQueue förlänger även gränssnittet IEnumerable, för att visa att det skall gå att iterera över kön.

Benchmarkprogrammet kommer alltid att köra en uppvärmningsomgång innan själva testerna börjar. Det denna uppvärmning gör är att den kör hela testet en gång men sparar inte resultaten till fil. Detta görs för att samtlig kod som används i testerna skall kompileras innan de faktiska testerna körs. Innan varje benchmark börjar kommer kön att initieras om och fyllas med initial data, sedan triggas en full garbage collection. Efter detta drar benchmarken igång och testerna utförs. Benchmarken börjar med att skapa de trådar den kommer att använda i testerna, de arbetstrådar som skapas får ett unikt ID. Detta ID börjar på ett och inkrementeras med varje ny

tråd. När alla trådar har skapats startas en Stopwatch (Microsoft u.å. e) som mäter hur läng tid benchmarken tar. När tidmätningen har börjat signaleras samtliga trådar att börja med sitt arbete, detta görs med hjälp av en boolesk flagga. Sedan får tråden som startat upp testerna sova så länge som det specificerats att testerna skall köra. När tråden vaknar signalerar den till samtliga trådar att avsluta sitt arbete, även detta görs med en boolesk flagga. Efter det väntas det tills samtliga trådar har gjort färdigt det arbete de håller på med, när det är gjort stoppas Stopwatchen och passerad tid avläses. Därefter väntas det in att samtliga trådar ska rapportera in sina resultat. När alla resultat rapporterats in sparas dessa ner till fil.

Varje arbetstråd som skapas sätter sin egen processoraffinitet, förutsatt att pinning inställningen är satt till alternate eller fill. När den är satt till fill sätts processoraffiniteten till trådens ID. Om den däremot är satt till alternate behövs mer beräkningar, för att komma fram till vilken processor tråden skall köras på. I följande formler representerar N antalet logiska kärnor som finns tillgängliga, i detta fall 24. För de (N/2) - 1 (iteratortråden tar en plats) första trådarna ska tråden köras på varje fysisk kärnas första logiska kärna, medan för de N/2 sista ska köras på varje fysisk kärnas andra logiska kärna. För de elva första trådarna blir då formeln som beräknar affiniteten med:

$$affinity_n = 2 * threadID$$

Affiniteten för de 12 sista trådarna beräknas då med formeln:

$$affinity_n = \left(threadID - \frac{N}{2}\right) * \frac{N}{2} + 1$$

Med hjälp av dessa formler fördelas alltså trådarna enligt Figur 4.2 – Figur 4.7. Utöver detta ansvarar även arbetstråden för att skapa sin egen slumpgenerator. För att få ett unikt frö i varje tråd används en kombination av tiden då programmet startade, antalet benchmarks som skall köras samt trådens id som frö. Denna slumpgenerator används sedan då tråden skall utföra en enqueue eller dequeue operation. Dessa operationer utförs med lika stor sannolikhet som slumpas inför varje operation.

Iteratortråden börjar med att sätta sin egen processoraffinitet, förutsatt att pinning inställningen är satt till alternate eller fill. Om iteratortråden själv skall avgöra vilken kärna den skall köras på kör den alltid på den första logiska kärnan. Därefter börjar iteratortråden att iterera över kön och gör detta så många gånger som möjligt fram till att den signaleras att avsluta.

Uppmätning av data

Nedan följer en beskrivning av hur benchmarkprogrammet mäter respektive mätdata. De olika mätdata som har uppmätts är följande:

- Lyckade och misslyckade enqueue samt dequeue operationer.
  Dessa antal räknas genom att en variabel inkrementeras varje gång en enqueue eller dequeue operation lyckas eller misslyckas inom varje tråd. Totalt finns det alltså fyra variabler inom varje tråd som representerar dessa. I slutet av varje benchmark adderas antalet lyckade och misslyckade operationer från samtliga trådar för att få det slutgiltiga värdena. Antalet misslyckade enqueue operationer kommer alltid vara noll eftersom ingen av de implementerade köerna har en enqueue operation som kan misslyckas. Benchmarkprogrammet implementerades så att det mäter denna siffra ändå ifall fler köer skulle läggas till i ett senare skede i studien, som medförde misslyckade enqueue operationer.

- Faktisk tid som benchmarken kör.
  För att mäta tiden experimenten kör har Stopwatch klassen från .NET biblioteket (Microsoft u.å. e) använts.

- Antalet iterationer som iterator tråden hinner genomföra.
  Det här antalet mäts genom att varje gång kön itereras inkrementeras en variabel.

- Antal luckor som påträffas i kön av iterationstråden.
  Denna data räknas genom att en variabel inkrementeras varje gång en lucka påträffas inom varje tråd. I slutet av varje benchmark adderas antalet påträffade luckor från samtliga trådar för att få det slutgiltiga värdet.

- Antalet element som itereras över i varje iteration.
  Detta beräknas genom att antalet itererade element varje iteration sparas. När benchmarken är färdig bearbetar den datan så att det blir en hanterbar mängd data som skrivs till fil. Det rör sig om stora mängder data som skall sparas i minnet under benchmarksen. Därför har storleken på datan beräknats för att vara säker på att minnesproblem inte skall uppstå.

4.2.2 Michael och Scott kön

Michael och Scott (1996) har framtagit den kö som är utgångspunkten vid implementation av Scan and Return och Double Collect iteratorerna. För att vara säkra på att läsning och skrivning av pekarna till huvud och svans har dessa pekare markerats som volatile.

4.2.3 Scan and Return

Scan and Return iteratorn, framtagen av Nikolakopoulos et al. (2013) är en iterator till Michael och Scott kön. Den bygger i sitt grundutförande som namnet antyder på att man först skannar av kön och sedan returnerar det tillstånd som sågs. Då det finns en garbage collector i den miljö experimenten är utförda i och kön utgörs av en länkad lista har dock inte hela kön behövts skannas av utan bara huvud och svans. Den hämtar först en pekare till huvudet och sedan en pekare till svansen, och returnerar sedan ett tillstånd byggt på dessa huvud- och svanspekare. Den har ingen kontroll på huruvida kön modifieras mellan det att huvudet hämtas och det svansen hämtas, alltså kan ingen garanti lämnas för att det tillstånd som returneras är ett atomiskt snapshot av kön.

Övergången från pseudo kod (Algoritm 1) till C# kod krävde endast några få modifikationer. Initieringen av StateToReturn (rad 3) ersätts med variabeldeklarationer för pekarna till huvud och svans. Läsningen av huvud- och svanspekarna (rad 4 & 5) kvarstår som de är. Returneringen av StateToReturn (rad 7) ersätts av en do while loop som itererar kön från den huvudpekare som lästs fram till den svanspekare som lästs, varje element som påträffas returneras med en yield return. Utöver detta så har en kontroll för om kön är tom lagts till mellan det att svansen hämtas och det att tillståndet returneras. Om kön är tom vid denna kontroll returnerar den direkt med en yield break.

4.2.4 Double Collect

Double Collect iteratorn, även den framtagen av Nikolakopoulos et al. (2013), är lik Scan and Return iteratorn. De skiljer sig från varandra på två punkter. Efter att svanspekaren har hämtats i Double Collect kontrolleras att huvudpekaren inte har ändrats i kön sedan denna hämtades. Det kontrolleras även att svanspekaren faktiskt pekar på sista elementet i kön. Dessa två

kontroller ger oss en garanti att det tillstånd som består av de hämtade huvud- och svanspekarna är ett atomiskt snapshot av kön, med synkroniseringspunkt där svanspekaren hämtas.

Även här var övergången till C# kod från pseudokod (Algoritm 2) simpel att göra. Initieringen av StateToReturn samt hämtningen av huvud- och svanspekare (rad 5-7) sker på samma sätt som i Scan and Return. Returneringen av StateToReturn (rad 10) flyttar vi ut utanför while loopen och ersätter den med en break sats. Själva returneringen sker på samma sätt som i StateToReturn, inklusive kontrollen för om kön är tom.

4.2.5 .NET Framework

Den parallella kö som finns tillgänglig i .NET, ConcurrentQueue (Microsoft u.å. a), har även testats. För att kunna testa den i benchmarkprogrammet som implementerats har en ny klass skapats. Denna klass agerar brygga, eller adapter, mellan framework kön och benchmarkprogrammet, vidarebefordrar alltså samtliga anrop.

4.2.6 Kö som bygger på ImmutableQueue

Som tidigare har tagits upp så används den ImmutableQueue (Microsoft u.å. d) som finns tillgänglig i .NET 4.5 och senare. Denna kö har använts för att representera en version av kön i en tillämpning av universella metoder för att konstruera parallella dataobjekt från sekventiella (Herlihy 1993; Anderson & Moir 1999). Kön som har implementerats har internt en referens till en ImmutableQueue som motsvarar köns nuvarande tillstånd. Vid operationer som modifierar köns tillstånd byts då istället detta tillstånd ut mot ett nytt innehållande de förändringar som utförts. Det tillstånd som det finns referens till kommer alltid att vara representativt för köns tillstånd vid en viss tidpunkt, och kommer aldrig att modifieras.

För att få kön att bli en ”normal” uppdateringsbar kö har nya enqueue, dequeue och iterator operationer implementerats. Pseudokod för dessa operationer finns i *3.2.3 Kön som bygger på Immutable*. Den nya implementation för enqueue börjar med att läsa köns nuvarande tillstånd, detta tillstånd kallas oldState. Efter det utförs en enqueue, där det tillstånd som returneras kallas för newState. Sedan sätts köns nuvarande tillstånd till newState om det fortfarande är oldState, detta görs med hjälp av en CAS. Om denna CAS misslyckas så görs ett nytt försök, detta förlopp upprepas tills att det lyckas. Dequeue operationen har även den implementerats på ett liknande sätt. Det börjar med att läsa köns nuvarande tillstånd, oldState. Sedan kontrolleras om oldState är en tom kö, om så är fallet returneras det att operationen misslyckats. Sedan dequeueas ett element från oldState och det tillstånds som returneras kallas för newState. Därefter sätts köns nuvarande tillstånd till newState om det fortfarande är oldState, detta görs med hjälp av en CAS. Om denna CAS misslyckas så görs ett nytt försök, detta förlopp upprepas tills det lyckas.

Eftersom immutable kön är säker att läsa från parallellt kan denna användas för att representera en version av kön på detta sätt. CAS operationenerna i enqueue och dequeue agerar minnesbarriär som gör att vi kan garantera att NewState är färdigskrivet till minnet innan någon annan tråd kan läsa den.

Iterationen sker genom att det nuvarande tillståndet läses och sedan returneras. Med tanke på att kön är uppbyggd på en immutable kö är det garanterat att det tillståndet som returneras aldrig kommer att förändras. Det garanterar även att det nuvarande tillståndet alltid representerar köns tillstånd vid en viss tidpunkt. Det ger en garanti att iteratorn alltid ger ett atomiskt snapshot av kön. Det leder till att modifikationer på kön efter att iteratorn hämtats inte återspeglas i iterationen.

4.3 Experiment

4.3.1 Experimentella förutsättningar

Testerna är utförda med en rad olika parameteruppsättningar. Samtliga tester startar upp N stycken trådar varav en är iterationstråd och resten är arbetstrådar. N antar värdena 2, 4, 8, 10, 12, 14, 16, 20, 22 och 24. Dessa arbetstrådar utför slumpmässigt enqueue och dequeue, med lika stor sannolikhet. Elementen arbetstrådarna sätter in i kön innehåller trådens ID samt ett sekvensnummer för just den tråden, med hjälp av denna data går det snabbt att upptäcka om det uppstår luckor av saknade element i kön. Iterationstråden itererar över kön så mycket den kan under experimentets gång. Den samlar samtidigt in data kring de element den itererar över. Antalet element som kön haft som initial storlek har varierat mellan är 0, 5 000 och 10 000. Varje experiment får köra i 20 sekunder, varefter samtliga trådar meddelas att avsluta sitt arbete. Trådarna avbryter alltså inte abrupt utan har tid på sig att avsluta sin nuvarande operation. Varje experiment upprepas 10 gånger. Samtliga utförda experiment har utförts på samtliga köer involverade i denna studie.

Som bekant så använder sig C# av JIT-kompilering. Detta tillför extra arbete som påverkar testtiderna. Detta bör undvikas för att få ett så opåverkat resultat som möjligt, därför körs varje experiment en extra gång före de övriga 10 körningarna. Den första körningens resultat varken uppmäts eller sparas undan. Även den garbage collector som C# använder sig av kan påverka våra testresultat. För att undvika detta i så stor mån som möjligt triggas manuellt en garbage collection före varje experiment.

4.3.2 Resultat från experiment

När benchmarkprogrammet kört klart testerna sparas resultaten till fil. Skrivningen till fil sker först efter att varje benchmarkkörning har kört klart för att denna inte skall påverka mättiderna. Medan benchmarken kör samlas det in data kring prestandan hos de olika operationerna hos kön. Antalet lyckade och misslyckade enqueue och dequeue operationer uppmäts. Antalet iterationer som hinns med räknas samt antalet element som itereras över i samtliga iterationer uppmäts. Antalet element som itereras över varje iteration skrivs dock inte till fil då detta hade blivit för mycket data att bearbeta i efterhand. Därför bearbetar benchmarkprogrammet denna data innan den skrivs till fil. Den data som skrivs till fil rörande antalet itererade element är lägsta och högsta antalet element som itereras över, medelvärdet samt antalet element som itererades över i sista iterationen. Iteratortråden räknar även antalet luckor som påträffas under samtliga iterationer. Utöver detta så mäts även den totala tid som benchmarken kör.

Totalt 6 olika parametrar har använts vid experimenten, varav 4 varierar. Dessa parametrar går att se i Tabell 2.1. Detta ger då totalt 360 stycken parameteruppsättningar. Experimenten har då producerat lika många textfiler med testresultat. Denna data har sammanfogats och grafer har producerats för att visualisera datan, och presenteras nedan.

4.3.3 Grafer

Nedan presenteras samtliga grafer som tagits fram från den data experimenten producerat. Tre olika typer av grafer har använts för att visualisera datan. Graferna presentas i nio grupper. Varje grupp representerar en kombination av pinning strategi och initial storlek på kön. X-axeln visar i samtliga grafer trådantal. Vad Y-axeln visar beskrivs i bildtexten. De fyra olika köernas namn är förkortade i graferna för att spara plats. Framework kön blir fw, kön som bygger på immutable blir im, Scan and Return blir sr och Double Collect blir dc. Grupperna är grupperade först på pinning strategi och sedan på initial storlek. Varje grupp innehåller dessa tre grafer:

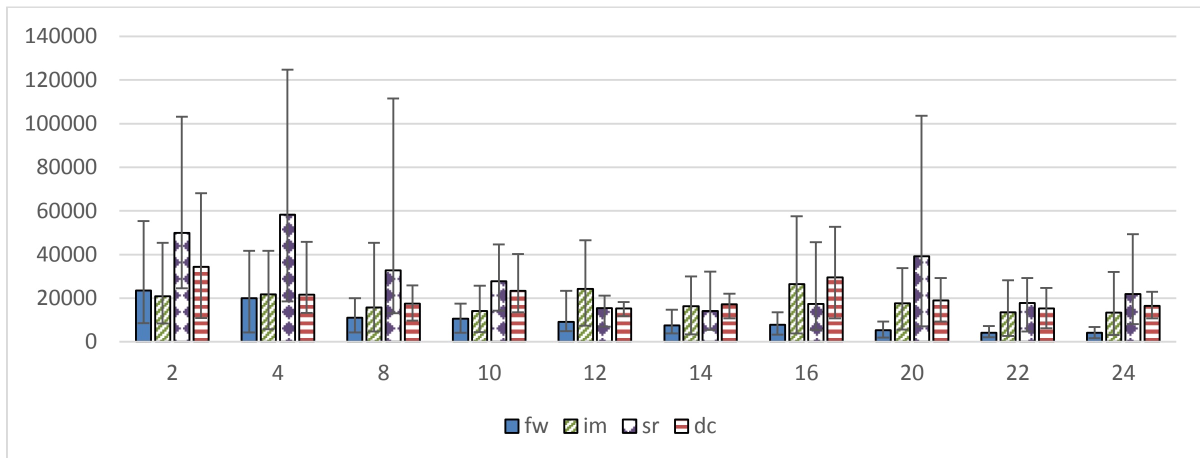
- Antal iterationer per sekund.
  Denna graf visar hur många iterationer iteratortråden hann med i genomsnitt per sekund under experimenten. Alltså hur många gånger iteratortråden hann hämta iteratorn samt iterera hela samlingen. Även högsta och lägsta antal iterationer visas.
  Exempel: Graf 4.1

- Genomsnittligt antal enqueue och dequeue operationer per sekund.
  Här presenteras hur många lyckade enqueue och dequeue operationer samtliga arbetstrådarna hann med i varje benchmark. Misslyckade dequeue operationer är alltså inte med i detta mått. Dessa presenteras som genomsnittligt antal per sekund.
  Exempel: Graf 4.2

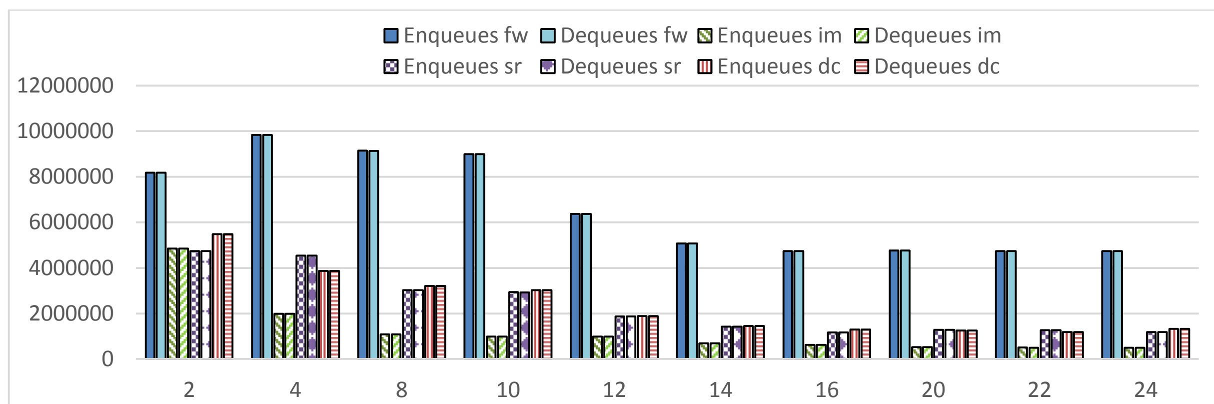
- Genomsnittligt antal element per iteration.
  I den här grafen presenteras ett genomsnitt över hur många element som iteratortråden itererat över i varje iteration. Detta ger ett ungefärligt mått på hur många element som befann sig i kön under experimenten.
  Exempel: Graf 4.3

Pinning strategin none

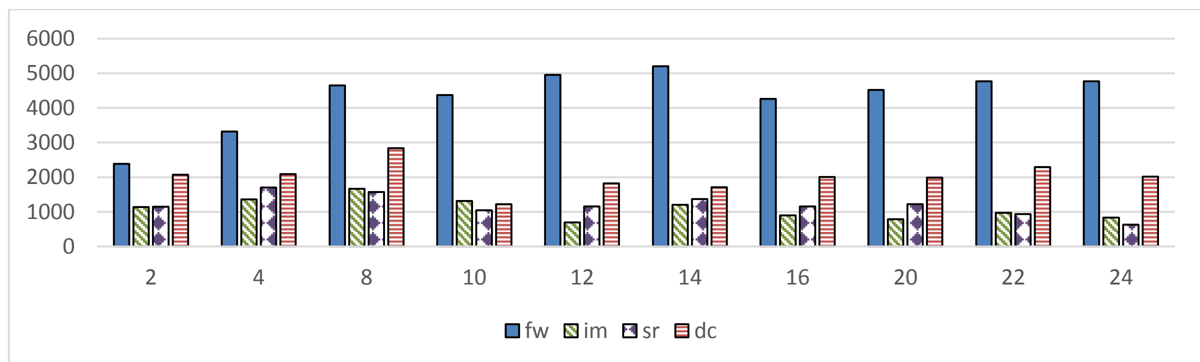
Nedan följer graferna för initial storlek 0 med pinning strategin none.



Graf 4.1 – Genomsnittligt antal iterationer per sekund med initial storlek 0 och pinning strategin none

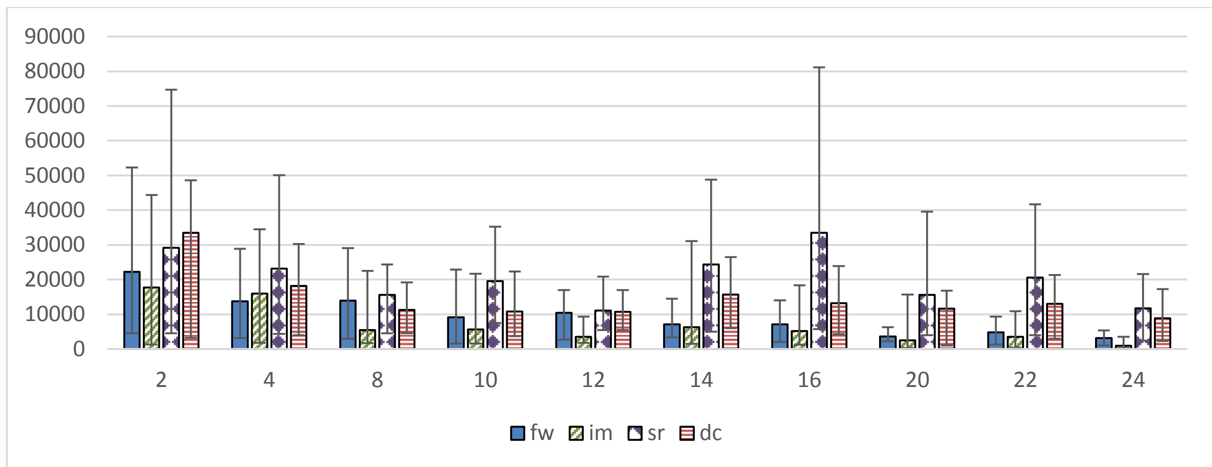


Graf 4.2 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek 0 och pinning strategin none

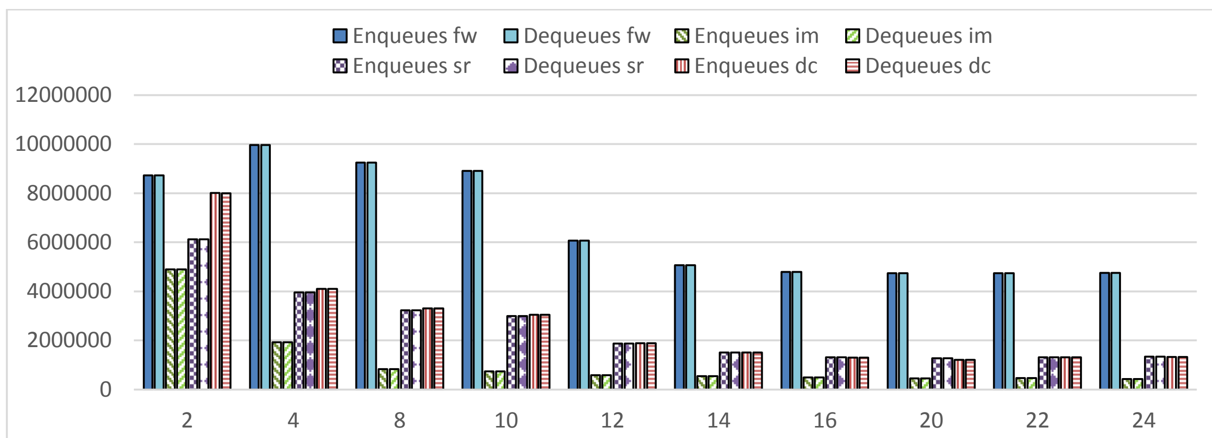


Graf 4.3 – Genomsnittligt antal element per iteration med initial storlek 0 och pinning strategin none

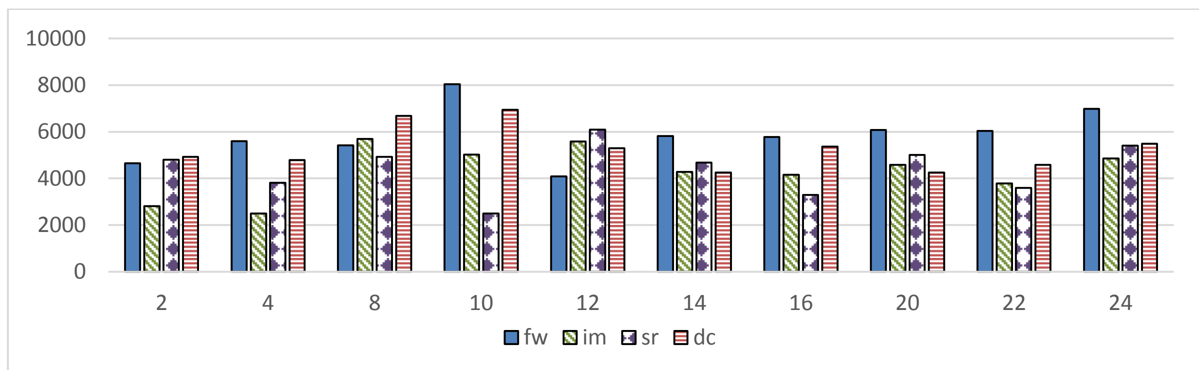
Nedan följer graferna för initial storlek 5 000 med pinning strategin none.



Graf 4.4 – Genomsnittligt antal iterationer per sekund med initial storlek på 5 000 och pinning strategin none

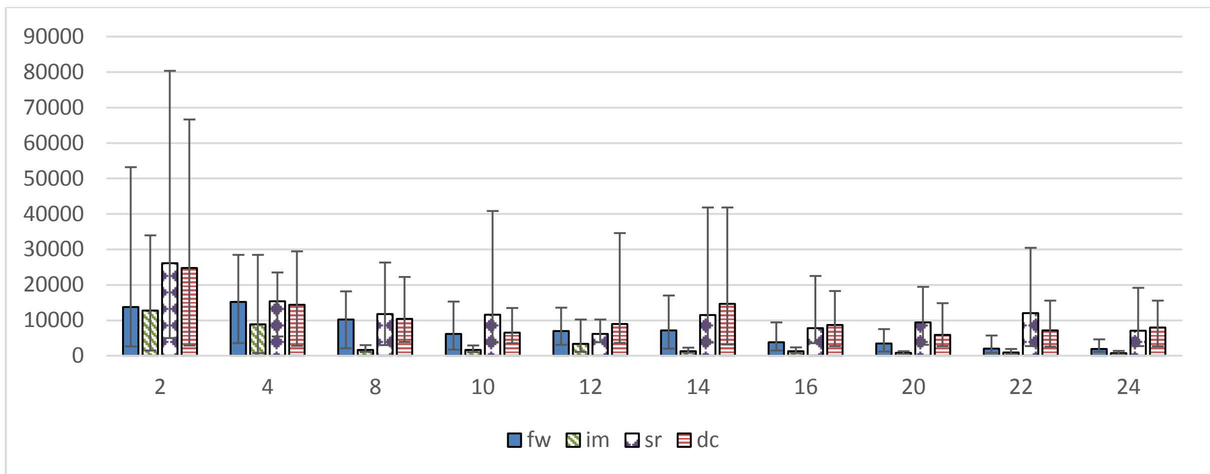


Graf 4.5 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek på 5 000 och pinning strategin none

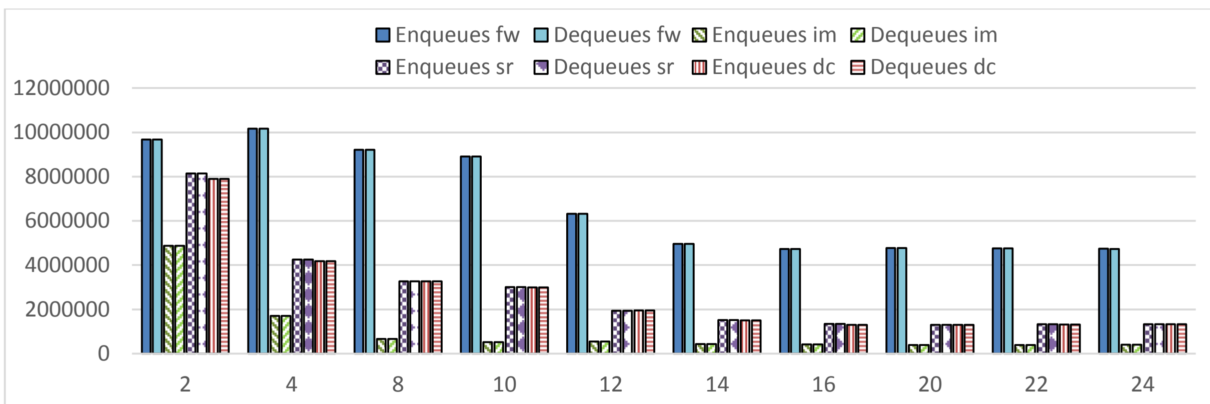


Graf 4.6 – Genomsnittligt antal element per iteration med initial storlek på 5 000 och pinning strategin none

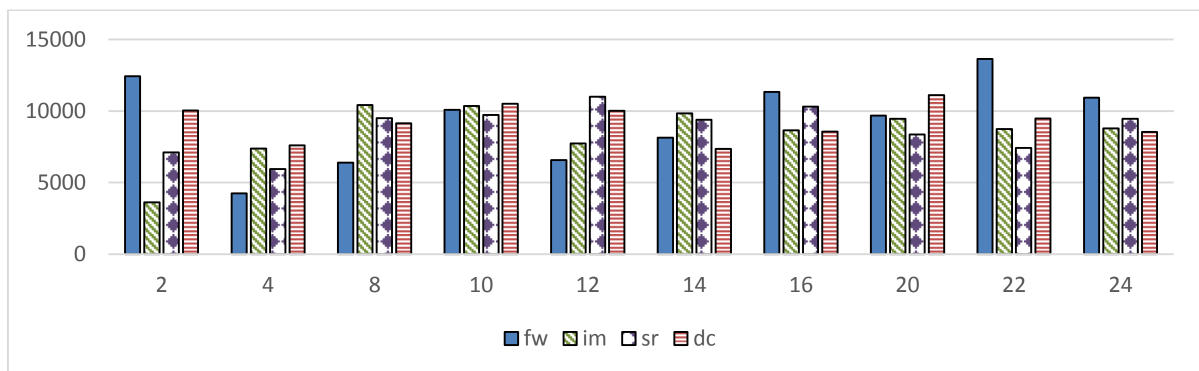
Nedan följer graferna för initial storlek 10 000 med pinning strategin none.



Graf 4.7 – Genomsnittligt antal iterationer per sekund med initial storlek 10 000 och pinning strategin none



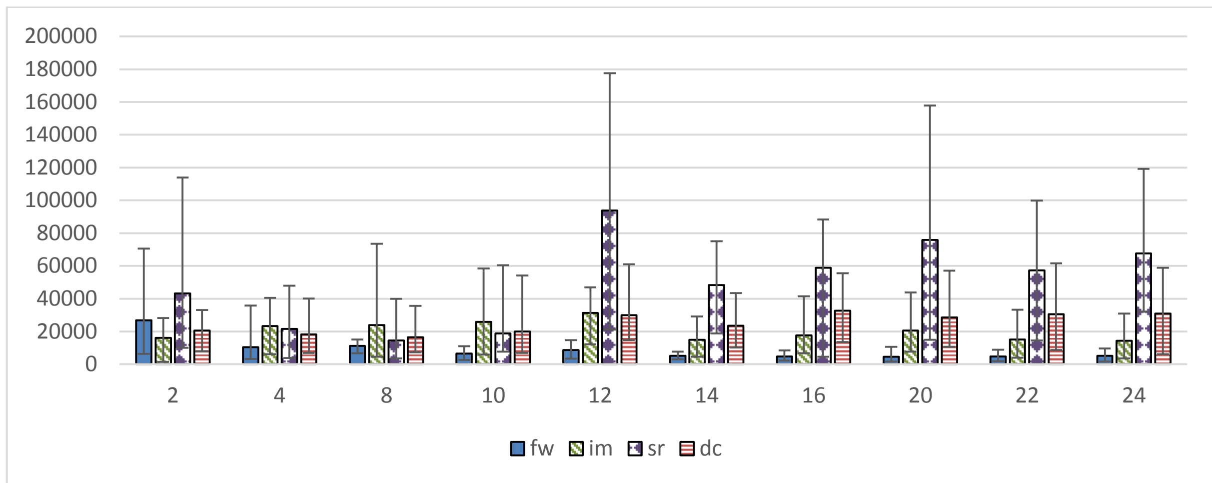
Graf 4.8 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek 10 000 och pinning strategin none



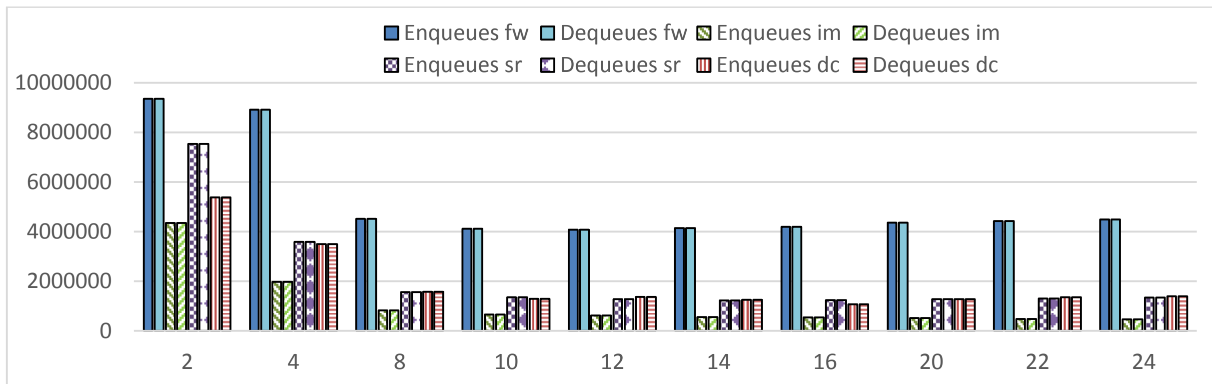
Graf 4.9 – Genomsnittligt antal element per iteration med initial storlek 10 000 och pinning strategin none

Pinning strategin alternate

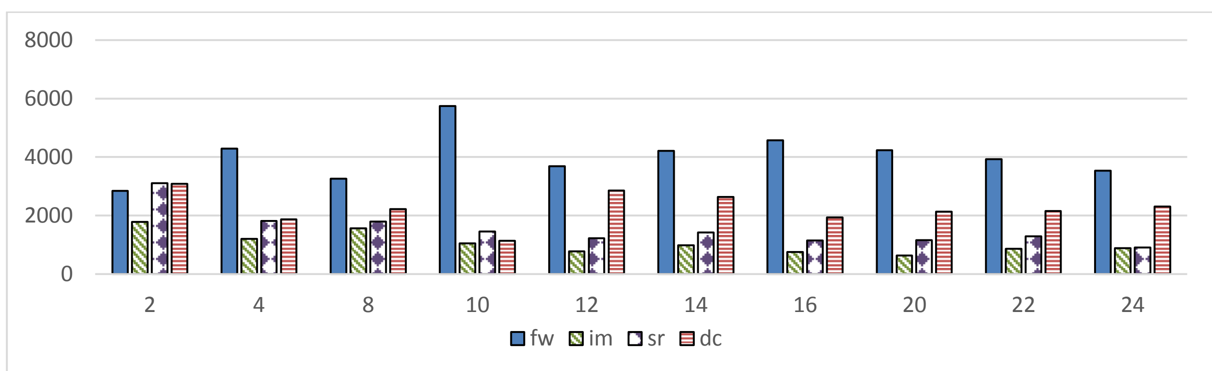
Nedan följer graferna för initial storlek 0 med pinning strategin alternate.



Graf 4.10 – Genomsnittligt antal iterationer per sekund med initial storlek 0 och pinning strategin alternate

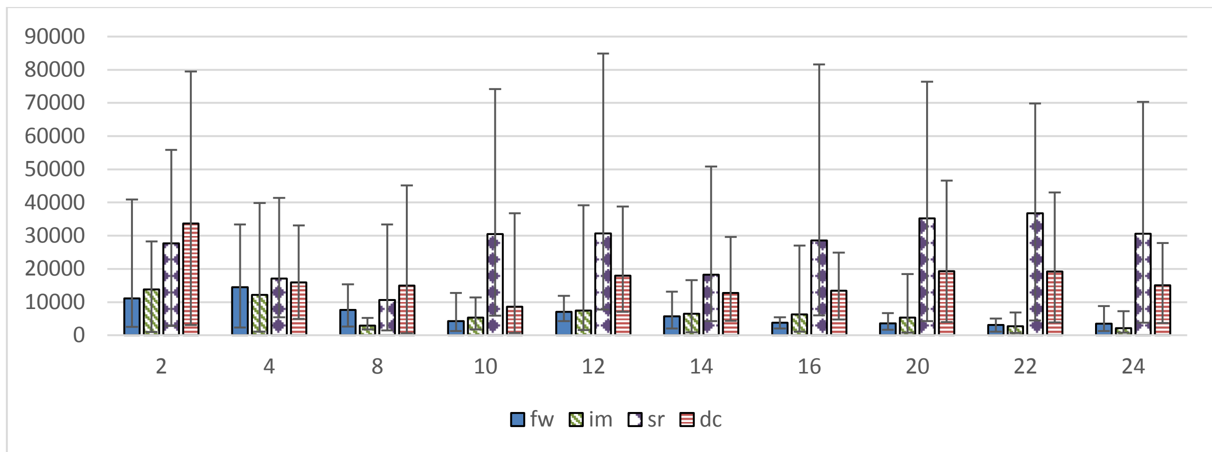


Graf 4.11 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek 0 och pinning strategin alternate

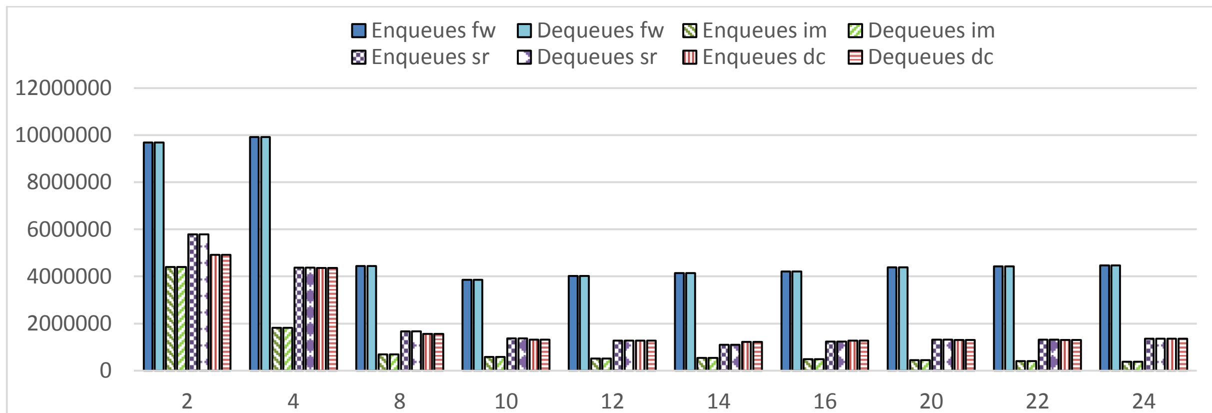


Graf 4.12 – Genomsnittligt antal element per iteration med initial storlek 0 och pinning strategin alternate

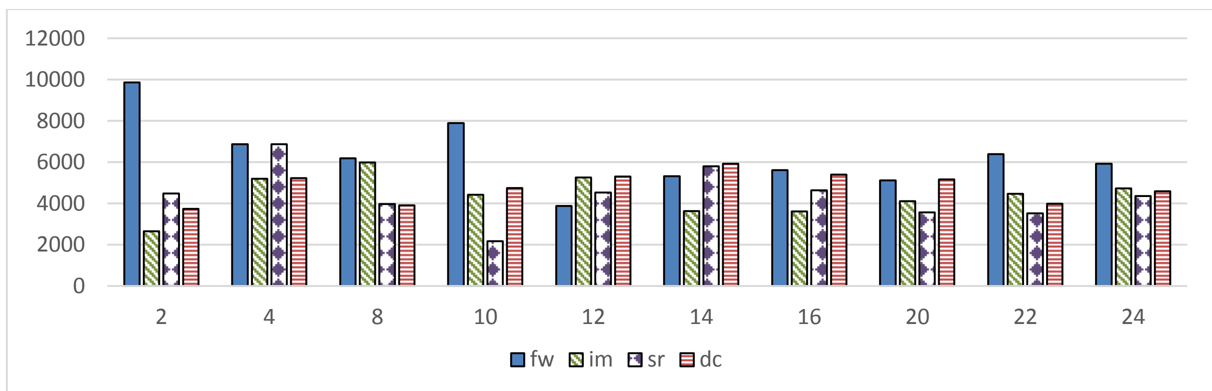
Nedan följer graferna för initial storlek 5 000 med pinning strategin alternate.



Graf 4.13 – Genomsnittligt antal iterationer per sekund med initial storlek på 5 000 och pinning strategin alternate

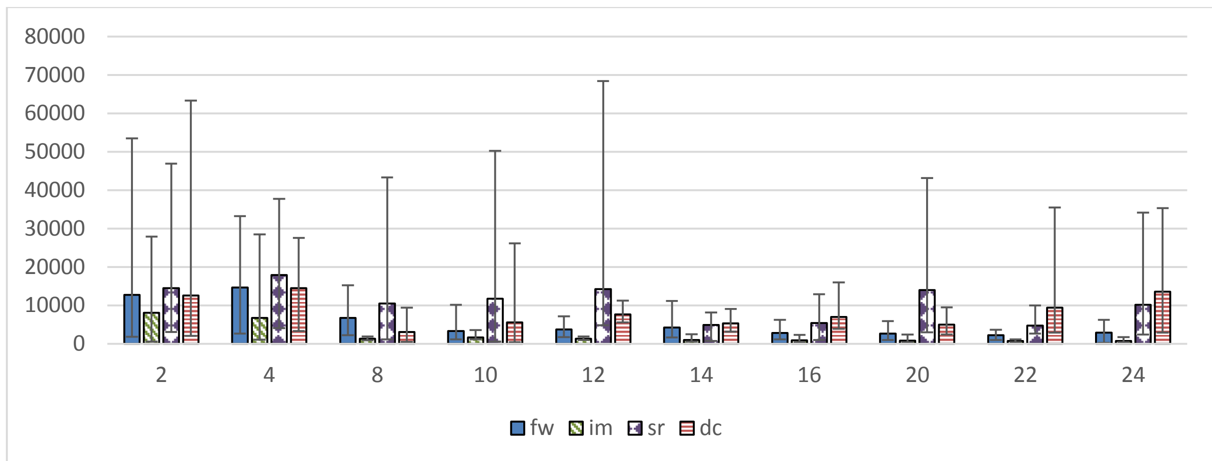


Graf 4.14 - Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek på 5 000 och pinning strategin alternate

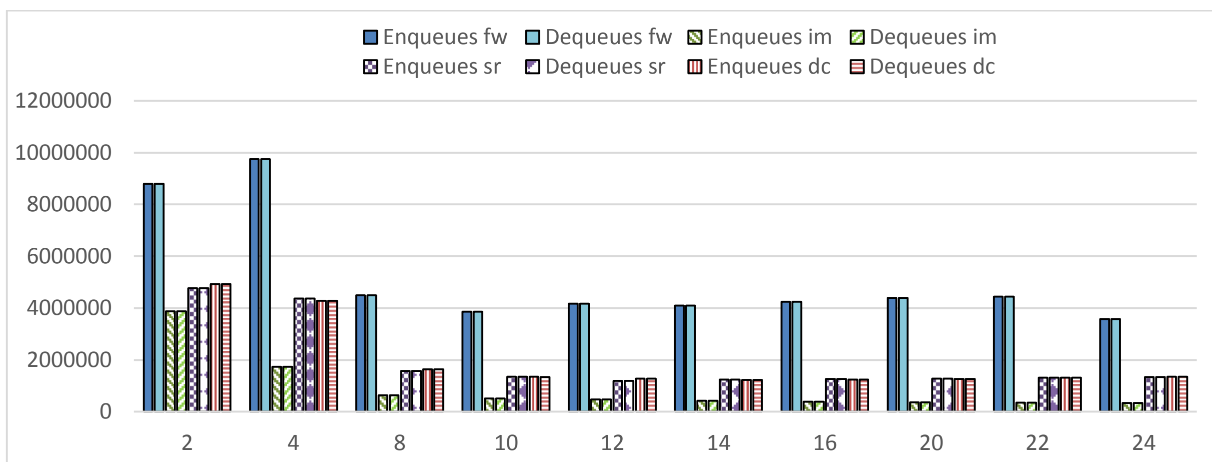


Graf 4.15 – Genomsnittligt antal element per iteration med initial storlek på 5 000 och pinning strategin alternate

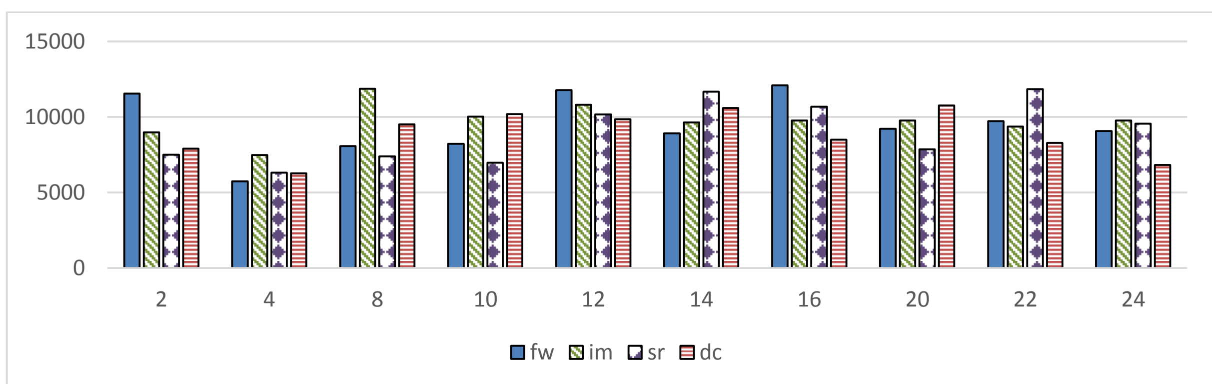
Nedan följer graferna för initial storlek 10 000 med pinning strategin alternate.



Graf 4.16 – Genomsnittligt antal iterationer per sekund med initial storlek 10 000 och pinning strategin alternate



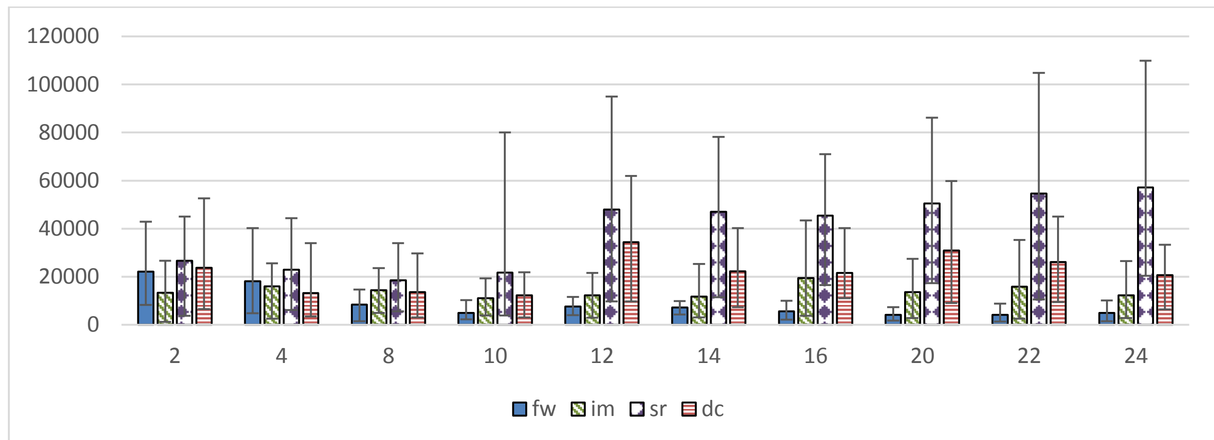
Graf 4.17 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek 10 000 och pinning strategin alternate



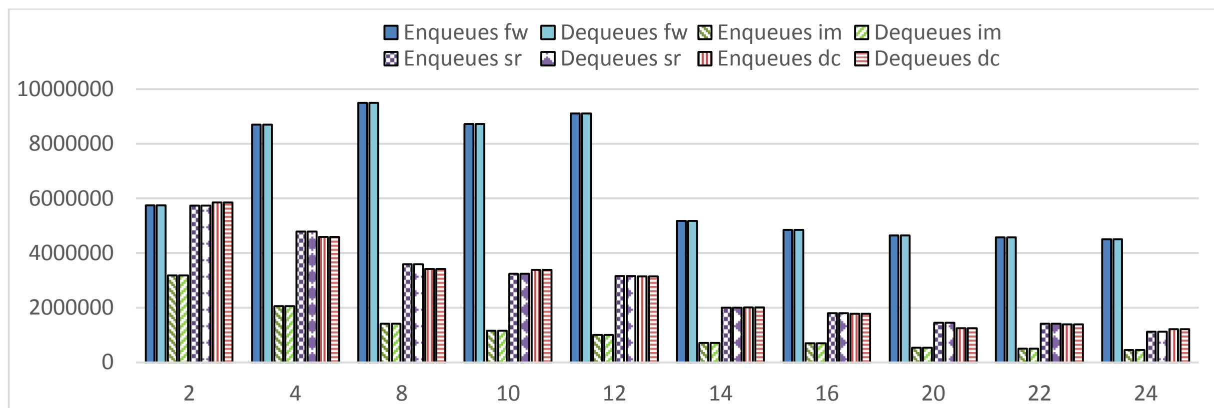
Graf 4.18 – Genomsnittligt antal element per iteration med initial storlek 10 000 och pinning strategin alternate

Pinning strategin fill

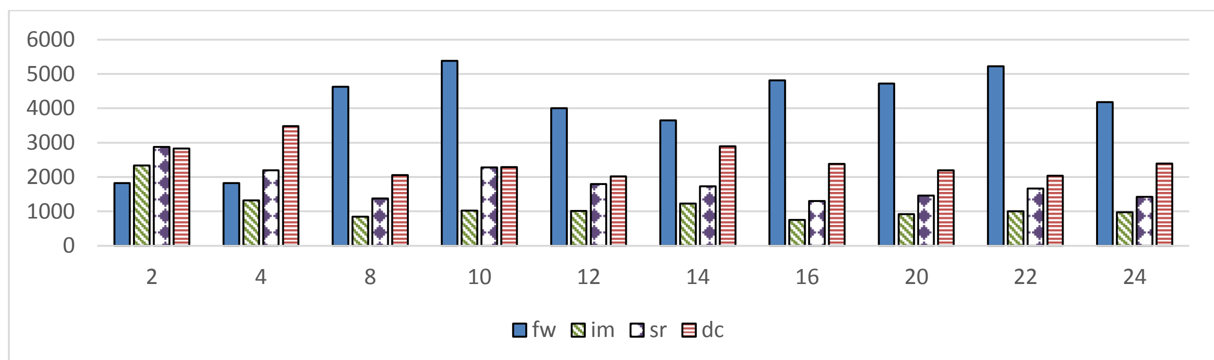
Nedan följer graferna för initial storlek 0 med pinning strategin fill.



Graf 4.19 – Genomsnittligt antal iterationer per sekund med initial storlek 0 och pinning strategin fill

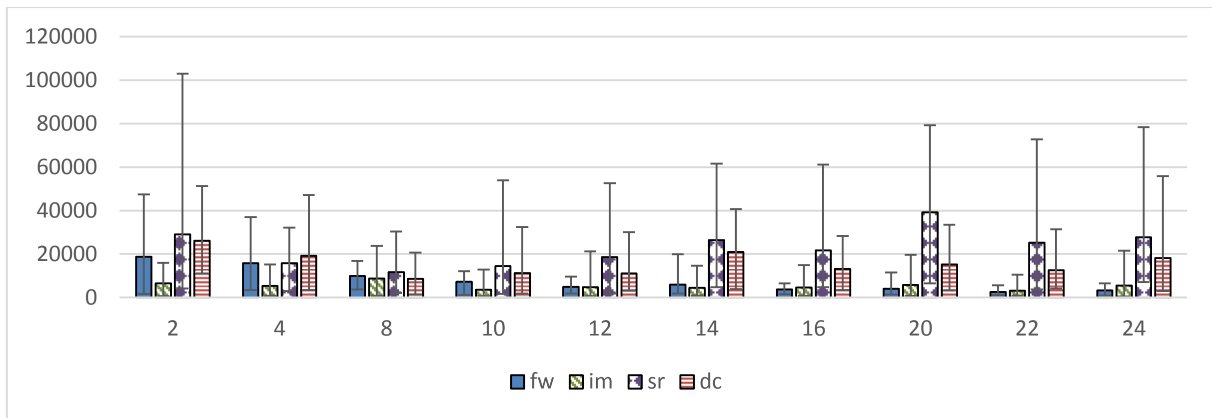


Graf 4.20 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek 0 och pinning strategin fill

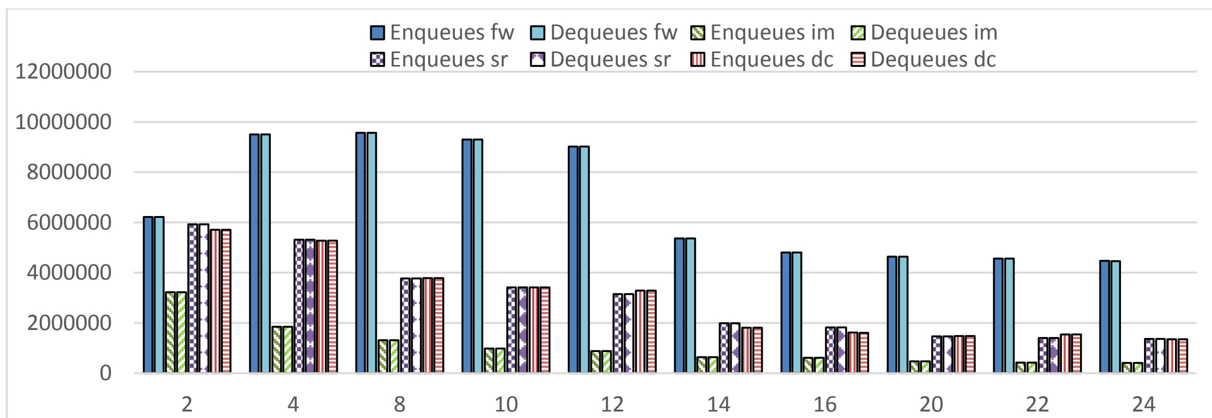


Graf 4.21 – Genomsnittligt antal element per iteration med initial storlek 0 och pinning strategin fill

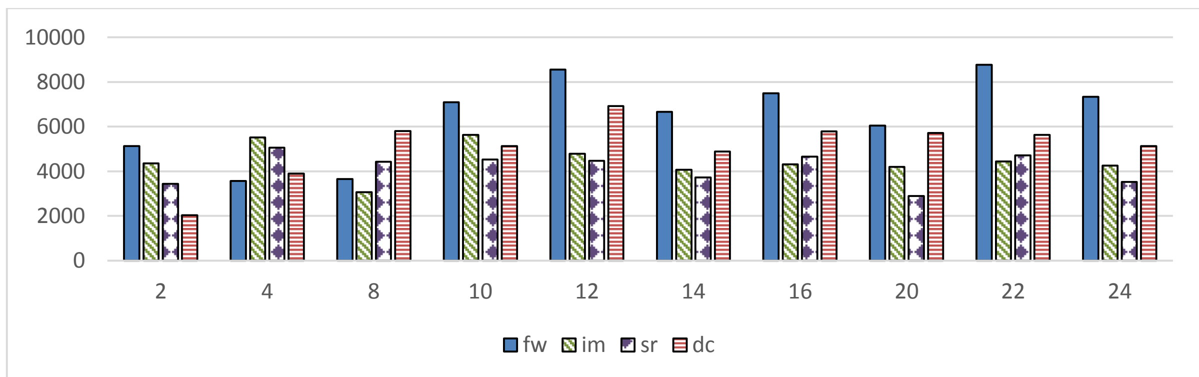
Nedan följer graferna för initial storlek 5 000 med pinning strategin fill.



Graf 4.22 – Genomsnittligt antal iterationer per sekund med initial storlek på 5 000 och pinning strategin fill

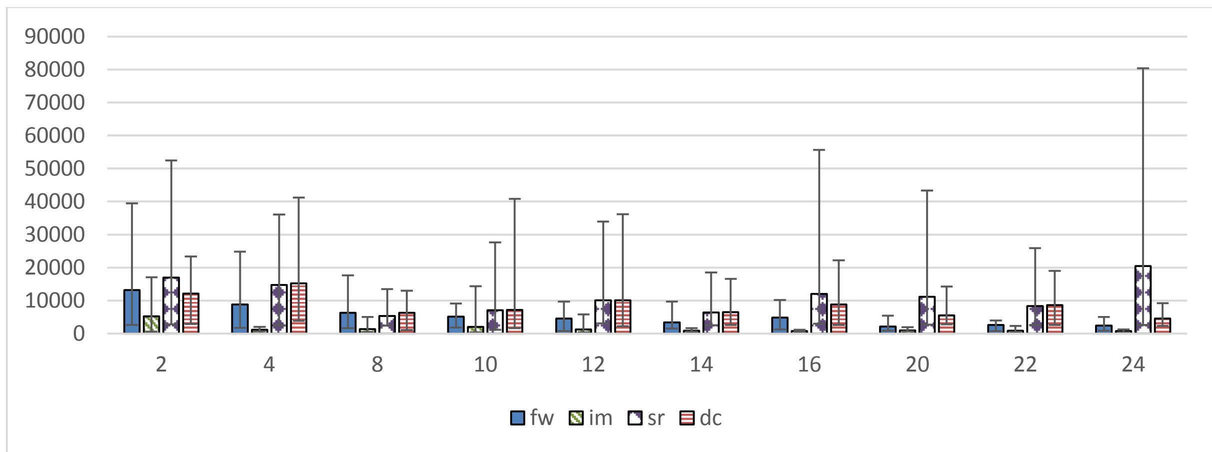


Graf 4.23 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek på 5 000 och pinning strategin fill

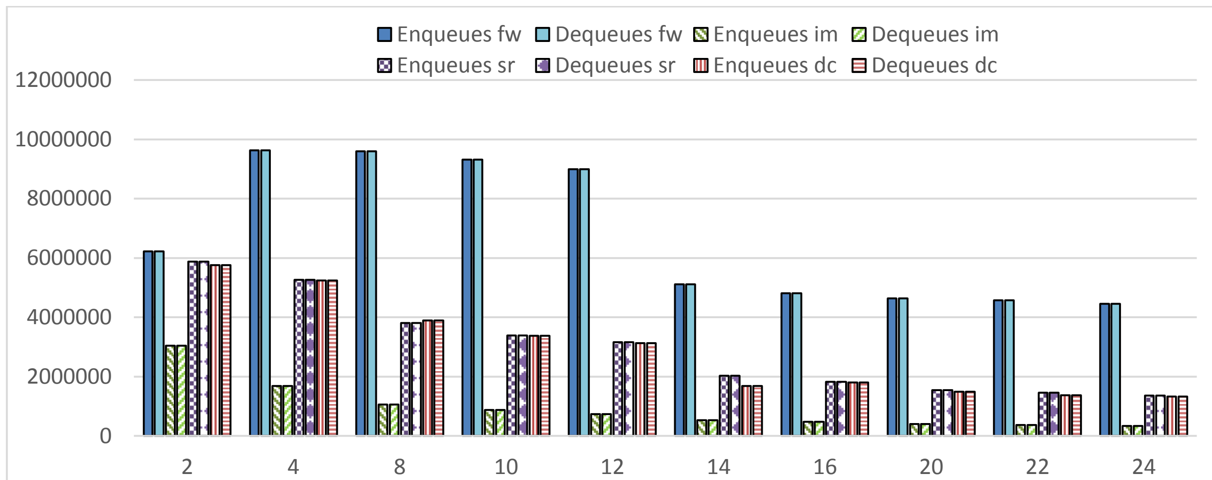


Graf 4.24 – Genomsnittligt element per iteration med initial storlek på 5 000 och pinning strategin fill

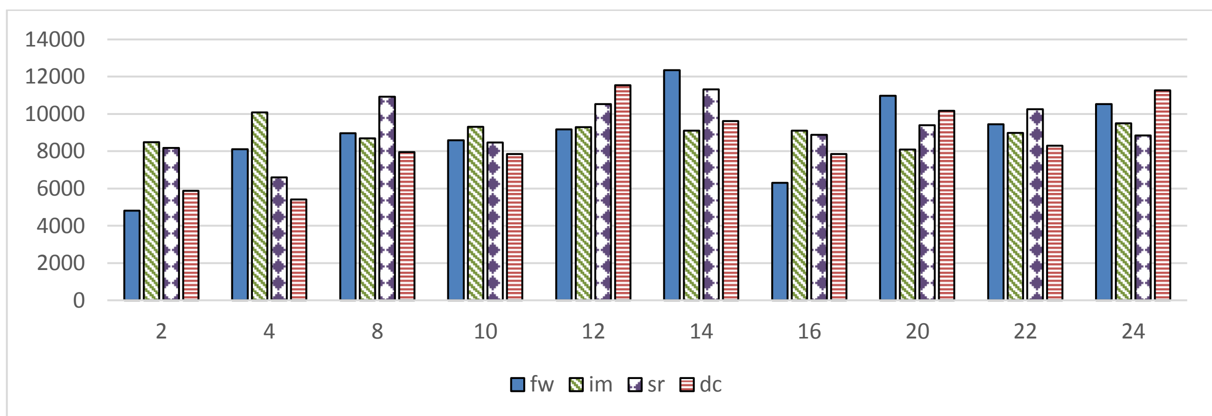
Nedan följer graferna för initial storlek 10 000 med pinning strategin fill.



Graf 4.25 – Genomsnittligt antal iterationer per sekund med initial storlek 10 000 och pinning strategin fill



Graf 4.26 – Genomsnittligt antal enqueue och dequeue operationer per sekund med initial storlek 10 000 och pinning strategin fill



Graf 4.27 – Genomsnittligt antal element per iteration med initial storlek 10 000 och pinning strategin fill

5 Analys

I det här kapitlet kommer resultaten från experimenten att analyseras. Först kommer påverkan av olika pinning strategier och initial storlek att diskuteras. Därefter kommer de olika köerna och dess iteratorer att jämföras mot varandra.

Som tidigare nämnts så har antalet luckor som påträffas under experimenten sparats. Det har dock inte uppstått några luckor under något av experimenten vilket bevisar att de köer som testats i experimenten inte tappar bort några element, samt att iteratorerna aldrig hoppar över något element.

Antalet itererade element per sekund visar sig variera relativt mycket från experiment till experiment. Detta beror helt enkelt på att detta mått är starkt bundet till antalet element som befinner sig i kön. Detta medför att det uppstår anomalier i vissa experiment för detta mått, där antalet element i kön har varierat mer åt något håll än normalt.

Den faktiska tiden som experimenten tog sparades även den ner. Vid analys av denna tid framgår det att experimenten ofta har en tid som är längre en de utsatta 20 sekunderna. Dock är tiden endast förlängd med upp mot 400 millisekunder i enstaka fall, medan medelvärdet ligger strax under 15 millisekunder. Vi har inte sett någon koppling mellan denna övertid och pinning strategier, trådantal eller initial storlek.

5.1 Pinning strategier

Det finns som väntat en del skillnader i prestandan vid användandet av olika pinning strategier. Inom de olika strategierna upptäcktes det oväntade skillnader mellan vissa trådantal. Vid jämförelserna av de olika pinning strategierna valdes en initial storlek på 5 000 element. Vid denna storlek finns potentialen för kön att bli tom under experimenten samtidigt som den inte är tom allt för ofta.

5.1.1 None

Pinning strategin none går ut på att låta operativsystemet hantera trådarna och bestämma vilken kärna dessa skall köras på. Som väntat går det att utläsa från Graf 4.4 att iteratorn hinner med flest iterationer när antalet trådar är som lägst. När trådantalet stiger blir det fler trådar som tävlar om processortid. Eftersom alla trådarna har samma prioritet och inte är satta till någon specifik kärna kan det antas att arbetstrådarna får mer processortid än iteratortråden då dessa är fler till antalet. Det är intressant att se att antalet iterationer ökar vid trådantalen 14 och 16. Denna anomali kan tänkas bero på antalet element i kön som iteratorn behöver iterera över, Graf 4.6 visar just detta. Det finns en tydlig koppling mellan antalet element i kön och antalet iterationer som hinns med. Framför allt för kön med Double Collect iteratorn syns det att när antalet element i kön sjunker vid 14 och 16 trådar stiger samtidigt antalet iterationer den hinner med. Även vid 22 trådar syns det att antalet element är lägre jämfört med 20 och 24 trådar, detta stämmer bra överens med att antalet iterationer ökar vid 22 trådar.

Det är också intressant att se att framework kön vid dessa parametrar inte är som snabbast på enqueue och dequeue (se Graf 4.5) vid 2 trådar, vilket alla övriga köer är. Istället har framework kön sitt maximum vid 4 trådar. Även vid 8 och 10 trådar är framework kön snabbare på enqueue och dequeue än vid 2 trådar.

5.1.2 Alternate

Pinning strategin alternate går ut på att fördela trådarna över varje fysisk kärna innan hypertrådningen börjar användas, se Figur 4.2 – Figur 4.4. I Graf 4.13 går det att se en markant

försämring av iteratorn vid just 8 stycken trådar. Det finns dock inget samband till antalet element i kön i det här fallet. Det går att se i Graf 4.15 att antalet element i kön inte stiger vid detta trådantal, det sjunker till och med för samtliga köer förutom den som baserats på en immutable kö. Det låga antalet iterationer vid 8 trådar går istället att förklara med att vid detta trådantal har trådar börjat läggas på processorn i den andra socketen. Till skillnad från trådantalet 4 då samtliga trådar ligger på processorn i den första socketen. Generellt sett går det att se en ökning i antalet iterationer som hinns med vid ökande trådantal. Detta skiljer sig från pinning strategin none där antalet iterationer minskade vid ökande trådantal. Detta kan tänkas bero på att iteratortråden nu har bundits till en egen kärna som ingen annan tråd använder. Alltså är det ingen annan tråd som tävlar om processortid på denna kärna och därmed får iteratortråden garanterat processortid på denna kärna. Även vid denna pinning strategi går det att se ett tydligt samband mellan antalet iterationer som hinns med och antalet element som finns i kön.

Man ser i Graf 4.14 att arbetstrådarna får ungefär lika mycket gjort vid 2 och 4 arbetstrådar. När antalet trådar däremot stiger till 8 så sjunker antalet enqueue och dequeue operationer som hinns med drastiskt, för samtliga köer, och ligger sedan på en jämn, smått stigande, nivå ändå upp till 24 trådar. Detta beror på att vid 4 trådar kör samtliga trådar på processorn i första socketen. När trådantalet går över sex så börjar trådar även läggas på processorn i den andra socketen.

Det är intressant att även vid denna pinning strategi så ökar antalet enqueue och dequeue operationer hos framework kön när vi går från 2 till 4 arbetstrådar, medan samtliga övriga köer minskar.

5.1.3 Fill

Pinning strategin fill går ut på att fylla processorn i första socketen innan trådar börjar läggas på processorn i andra socketen, se Figur 4.5 – Figur 4.7. Vid denna pinning strategi går det att utläsa i Graf 4.22 att iteratorn generellt sett hinner med fler iterationer desto fler trådar som används, likt pinning strategin alternate. Även här kan det tänkas bero på att iteratortråden bundits till en egen kärna och därmed garanterat får processortid på denna. Denna ökning kan även tänkas bero på att arbetstrådarna hinner med mindre arbete desto fler trådar som används, se Graf 4.23. Desto mindre arbete arbetstrådarna lyckas genomföra, desto större fönster ges iteratortråden att hämta iteratorn utan att kön modifieras av någon arbetstråd.

Som väntat går det i Graf 4.23 att se en stadig minskning av antalet enqueue och dequeue operationer som hinns med, med en tröskel mellan 12 och 14 trådar. Detta beror på att det är här som processorn i den andra socketen börjar användas.

Även vid denna pinning strategin ser vi en tydlig ökning i antalet enqueue och dequeue operationer utförda av framework kön när vi går från 2 till 4 stycken trådar.

5.1.4 Sammanfattning av pinning strategier

Då antalet iterationer som hinns med är starkt bundet till antalet element i kön är det svårt att använda iteratorns prestanda som jämförelsemått mellan de olika pinning strategierna som testats. Arbetstrådarnas prestanda passar bättre till detta ändamål då den inte påverkas av antalet element i kön. Denna data finns visualiserad i Graf 4.5, Graf 4.14 och Graf 4.23.

Pinning strategin alternate är den strategi som i experimenten visat sig vara sämst. Vid samtliga trådantal presterar den sämre än de andra två testade strategierna. När trådantalet ökar så närmar den sig de andra strategierna och når ungefär samma nivåer vid 24 trådar. Strategin none visar sig vara något bättre än fill vid 2 trådar. Detta är trovärdigt eftersom Windows då kan sprida ut

trådarna på fler kärnor för att utnyttja alla tillgängliga kärnor bättre än vad fill strategin kan. Vid 12 stycken trådar presterar fill tydligt bättre än none. Även vid 14 trådar är fill bättre än none men inte lika markant som vid 12 trådar. Vid övriga trådantal presterar dessa strategier ungefär lika bra. Anledningen att strategin fill är bättre än none vid just 12 trådar beror på att det är vid just detta trådantal som fill lägger samtliga trådar på processorn i första socketen, medan none antagligen sprider ut trådarna jämt på processorerna i båda socketsena.

Det går även att observera i Graf 4.5, Graf 4.14 och Graf 4.23 att strategin fill är något sämre vad gäller enqueue och dequeue operationer än både none och alternate vid 4 trådar. Detta beror på att fill här utnyttjar hypertrådning på två stycken fysiska kärnor, medan alternate och antagligen också none delar ut trådarna på var sin fysisk kärna istället.

Värt att nämna är att många av de skillnader mellan olika pinning strategier som uppstod i experimenten kan vara beroende av hårdvara. Experimenten har trots allt körts på en dator med två stycken processorer. Om experimenten hade körts på en dator med endast en processor hade skillnaderna mellan pinning strategierna antagligen varit annorlunda.

5.2 Initial storlek

Tabellerna nedan, Tabell 5.1, Tabell 5.2 och Tabell 5.3, visar hur många dequeue operationer som misslyckats i genomsnitt för varje parameteruppsättning. Med tanke på hur många operationer arbetstrådarna hinner med under experimenten är det förvånansvärt få dequeue operationer som misslyckats. Framework kön till exempel gör mellan strax under fyra miljoner upp till närmare tio miljoner dequeue operationer per sekund, i experiment som kör under 20 sekunder. Att det som högst misslyckas lite mer än 20 000 dequeue operationer under hela experimentet är en förvånansvärt låg siffra. Det framgår i tabellerna att antalet misslyckade dequeue operationer minskar när den initiala storleken på kön ökar. Detta är inte förvånande eftersom en misslyckad dequeue endast kan uppstå när kön är tom och inget mer element kan tas bort. Denna trend är närvarande för samtliga köer samt samtliga pinning strategier. Det är väntat att detta mönster uppstår då det är för att undvika att kön blir tom som den initiala storleken varierats.

Som det går att utläsa i tabellerna nedan är det framework kön som misslyckas med flest dequeue operationer. Scan and Return och Double Collect ligger på ungefär samma nivå, Double Collect misslyckas med några färre. Detta är inte förvånande då de bygger på samma grundimplementation och Double Collect iteratorn hjälper svansen hinna ikapp om den skulle halka efter, vilket kan tänkas snabba upp enqueue och dequeue operationerna något. Den kö som bygger på en immutable kö har minst antal misslyckade dequeue operationer. Det är inte oväntat eftersom denna kö även är den som utför minst antal dequeue operationer. Detta går att se i den andra grafen inom varje grupp, exempelvis Graf 4.2.

Pinning: None								
Initial size: 0								
	fw	%	im	%	sr	%	dc	%
2	12491,2	0,0076%	10913,2	0,0112%	14972,7	0,0158%	10936,3	0,0100%
4	20112,4	0,0102%	6041	0,0152%	11790,8	0,0130%	8142,2	0,0105%
8	12121,7	0,0066%	4851,8	0,0223%	11982	0,0198%	7311,8	0,0114%
10	11950,1	0,0066%	4451,8	0,0225%	10227,7	0,0174%	11460,2	0,0189%
12	9386,5	0,0074%	9158,5	0,0463%	4375,7	0,0117%	5402,8	0,0143%
14	8048,3	0,0079%	3463,3	0,0249%	4242,8	0,0148%	4760,8	0,0164%
16	12261,5	0,0129%	5197,2	0,0420%	4305,5	0,0183%	5643,8	0,0217%
20	10999,5	0,0115%	3675,4	0,0355%	7949,1	0,0311%	6168,5	0,0245%
22	7966,2	0,0084%	3256,6	0,0323%	6005,3	0,0237%	4188,8	0,0176%
24	11019,5	0,0116%	3391,2	0,0341%	7262,6	0,0304%	4557,9	0,0172%
Initial size: 5 000								
	fw	%	im	%	sr	%	dc	%
2	9181,4	0,0053%	9114	0,0093%	6880,3	0,0056%	10995,6	0,0069%
4	10812,7	0,0054%	4499,8	0,0117%	7710	0,0097%	6483,1	0,0079%
8	15224,2	0,0082%	986,9	0,0059%	3610,4	0,0056%	1333,1	0,0020%
10	12015,8	0,0067%	1057,8	0,0072%	5028,5	0,0084%	2340,8	0,0038%
12	13787,4	0,0113%	223,4	0,0019%	1873,4	0,0050%	1605,1	0,0043%
14	5869,8	0,0058%	979,7	0,0091%	1938,1	0,0064%	1393,9	0,0046%
16	9048,3	0,0094%	515,3	0,0053%	3370,1	0,0128%	736,5	0,0028%
20	3890,7	0,0041%	212,3	0,0024%	3296,2	0,0130%	1082,1	0,0045%
22	9882	0,0104%	813,2	0,0089%	3856,5	0,0147%	2775,3	0,0106%
24	5447,5	0,0057%	348,3	0,0041%	1090	0,0041%	1179,4	0,0044%
Initial size: 10 000								
	fw	%	im	%	sr	%	dc	%
2	5810,3	0,0030%	5647,4	0,0058%	5674,3	0,0035%	7480,7	0,0047%
4	13578,9	0,0067%	2602,9	0,0077%	3403,2	0,0040%	3710,7	0,0044%
8	8201,1	0,0044%	0	0,0000%	2204,3	0,0034%	1739,6	0,0027%
10	5528,9	0,0031%	0	0,0000%	1540,9	0,0026%	307,5	0,0005%
12	6167,1	0,0049%	530,3	0,0048%	132,1	0,0003%	688,2	0,0018%
14	9086,9	0,0092%	0	0,0000%	534,1	0,0018%	2033,6	0,0068%
16	2897,8	0,0031%	0	0,0000%	109,8	0,0004%	335,7	0,0013%
20	4769,1	0,0050%	0	0,0000%	838,2	0,0032%	339,4	0,0013%
22	1043,5	0,0011%	0	0,0000%	1468,8	0,0056%	499,5	0,0019%
24	1554,6	0,0016%	0	0,0000%	421,8	0,0016%	243,1	0,0009%

Tabell 5.1 – Genomsnittligt antal misslyckade dequeue operationer med pinning strategin none

Pinning: Alternate								
Initial size: 0								
	fw	%	im	%	sr	%	dc	%
2	14436,3	0,0077%	8851,3	0,0102%	12518,1	0,0083%	5245,6	0,0049%
4	9522,8	0,0053%	8315,9	0,0211%	10689,6	0,0149%	9008,9	0,0129%
8	10803,7	0,0119%	5447,4	0,0331%	3364,7	0,0107%	3919,4	0,0124%
10	7529,4	0,0091%	3664,4	0,0280%	3513	0,0130%	7385,9	0,0284%
12	13782	0,0169%	4490,6	0,0358%	8396,9	0,0329%	4738,9	0,0173%
14	10160,8	0,0123%	3116	0,0279%	4724,8	0,0192%	3787,8	0,0151%
16	10484,4	0,0125%	4175,6	0,0383%	5668,3	0,0228%	5093,7	0,0238%
20	10242,5	0,0117%	4959,7	0,0475%	9139,2	0,0355%	5433,1	0,0213%
22	10914,7	0,0123%	2991,7	0,0313%	5529,8	0,0212%	5945,6	0,0220%
24	12453,8	0,0138%	3238,6	0,0344%	8600,7	0,0319%	6876	0,0245%
Initial size: 5 000								
	fw	%	im	%	sr	%	dc	%
2	3929,5	0,0020%	6520,3	0,0074%	7672,8	0,0066%	11581,2	0,0118%
4	10476,2	0,0053%	4023,8	0,0111%	4527,2	0,0052%	5337,5	0,0061%
8	4261	0,0048%	0	0,0000%	2998,7	0,0090%	2865,4	0,0092%
10	5244,1	0,0068%	228	0,0020%	3591	0,0131%	1590,8	0,0060%
12	8416,9	0,0105%	539,2	0,0053%	1554,7	0,0061%	1110,3	0,0044%
14	9112,4	0,0110%	588,6	0,0055%	560,7	0,0025%	950,9	0,0039%
16	3538,6	0,0042%	912,5	0,0094%	2093	0,0085%	595,3	0,0023%
20	6246,5	0,0071%	888,3	0,0101%	2910,6	0,0110%	2775,4	0,0106%
22	4030,3	0,0045%	236,9	0,0029%	3158,4	0,0120%	3325,6	0,0128%
24	5438,3	0,0061%	109,4	0,0014%	2818,3	0,0104%	993,1	0,0036%
Initial size: 10 000								
	fw	%	im	%	sr	%	dc	%
2	5768,8	0,0033%	4447,7	0,0057%	3016	0,0032%	2910,1	0,0030%
4	11321,4	0,0058%	1440,8	0,0042%	4131,4	0,0047%	4760,7	0,0056%
8	5857,1	0,0065%	0	0,0000%	762,1	0,0024%	1120,5	0,0034%
10	3255,1	0,0042%	0	0,0000%	1991	0,0074%	319,6	0,0012%
12	446,3	0,0005%	0	0,0000%	544,8	0,0023%	0	0,0000%
14	6891,9	0,0084%	0	0,0000%	0	0,0000%	0	0,0000%
16	2262,9	0,0027%	0	0,0000%	0	0,0000%	340,6	0,0014%
20	2828,5	0,0032%	0	0,0000%	708,2	0,0028%	0	0,0000%
22	2032	0,0023%	0	0,0000%	0	0,0000%	1035,2	0,0039%
24	4178	0,0047%	0	0,0000%	343,6	0,0013%	2026,7	0,0075%

Tabell 5.2 – Genomsnittligt antal misslyckade dequeue operationer med pinning strategin alternate

Pinning: Fill								
Initial size: 0								
	fw	%	im	%	sr	%	dc	%
2	10491,3	0,0091%	8707	0,0137%	8672,7	0,0076%	8886,9	0,0076%
4	23721,1	0,0136%	8540,3	0,0207%	10155,8	0,0106%	7735,3	0,0084%
8	16209,8	0,0085%	6858,3	0,0242%	10980	0,0153%	9103,4	0,0133%
10	11225,1	0,0064%	4906,4	0,0212%	6700,7	0,0103%	9876,3	0,0146%
12	14678,7	0,0080%	5289,6	0,0264%	10884,6	0,0172%	12110,3	0,0192%
14	10015,6	0,0097%	3177,7	0,0223%	7536,5	0,0188%	5751,2	0,0143%
16	11461,6	0,0118%	6581,4	0,0471%	6503,1	0,0181%	6045,1	0,0170%
20	9363,8	0,0101%	3574,6	0,0331%	5430	0,0187%	6154,3	0,0246%
22	7919,6	0,0086%	3826,1	0,0380%	6897,5	0,0243%	4829,6	0,0174%
24	13472,2	0,0149%	3004,8	0,0329%	5288,7	0,0236%	3497,8	0,0130%
Initial size: 5 000								
	fw	%	im	%	sr	%	dc	%
2	9814	0,0079%	3325,6	0,0052%	10386,5	0,0088%	8836,5	0,0077%
4	15518,7	0,0082%	1990	0,0054%	4439,2	0,0042%	8680,3	0,0082%
8	13907,9	0,0073%	3635,8	0,0138%	5641,6	0,0075%	4319,3	0,0057%
10	10851,2	0,0058%	968,8	0,0049%	5492,6	0,0080%	5274,1	0,0077%
12	6676,9	0,0037%	1456,4	0,0083%	3053,6	0,0048%	2582,8	0,0039%
14	6480,8	0,0060%	858,5	0,0067%	4459,1	0,0112%	5071,9	0,0140%
16	5500,3	0,0057%	985,9	0,0081%	1914,9	0,0052%	2366,7	0,0073%
20	7061,3	0,0076%	1013,2	0,0107%	3763,7	0,0129%	2648,8	0,0090%
22	4304,1	0,0047%	276	0,0033%	1817,9	0,0065%	895,5	0,0029%
24	5918,9	0,0066%	1053,8	0,0130%	2061,8	0,0075%	2307,9	0,0085%
Initial size: 10 000								
	fw	%	im	%	sr	%	dc	%
2	6288,2	0,0050%	2854,9	0,0047%	5354,7	0,0046%	3539,2	0,0031%
4	6745,1	0,0035%	0	0,0000%	4887,9	0,0046%	7694,2	0,0073%
8	8237,5	0,0043%	160,9	0,0008%	676,5	0,0009%	2068,2	0,0027%
10	5717,4	0,0031%	460,1	0,0026%	1998,6	0,0029%	1812,3	0,0027%
12	7719,9	0,0043%	177,5	0,0012%	1248,2	0,0020%	1727,7	0,0028%
14	2573,5	0,0025%	0	0,0000%	38,3	0,0001%	166,8	0,0005%
16	8190,1	0,0085%	0	0,0000%	1104,4	0,0030%	547,1	0,0015%
20	2300,2	0,0025%	0	0,0000%	522,8	0,0017%	124,2	0,0004%
22	3473,2	0,0038%	0	0,0000%	430,5	0,0015%	623,1	0,0023%
24	2668,2	0,0030%	0	0,0000%	2223,1	0,0081%	0	0,0000%

Tabell 5.3 – Genomsnittligt antal misslyckade dequeue operationer med pinning strategin fill

Total				
	fw	im	sr	dc
Max	0,0169%	0,0475%	0,0355%	0,0284%
Avg	0,0070%	0,0123%	0,0100%	0,0085%
Min	0,0005%	0,0000%	0,0000%	0,0000%

Tabell 5.4 – Högsta, minsta och medelvärde för andelen misslyckade dequeue operationer

Som Tabell 5.4 visar är det inte så att det är framework kön som misslyckas med störst andel av sina dequeue operationer. Bortsett från det faktum att den aldrig når noll misslyckade dequeue operationer är detta den kö som misslyckas med minst antal dequeue operationer i samtliga fall. Den kö som bygger på en immutable kö visar sig vara den kö som misslyckas med störst andel av sina dequeue operationer. Scan and Return misslyckas med mindre andel av sina dequeue operationer än den kö som bygger på immutable. Double Collect i sin tur misslyckas med mindre andel av sina dequeue operationer än Scan and Return. Det går att se ett tydligt samband mellan andelen misslyckade dequeue operationer och antalet utförda enqueue och dequeue operationer. Desto fler enqueue och dequeue operationer som utförs, desto mindre andel av dequeue operationerna misslyckas. Denna trend är närvarande för samtliga köer.

5.3 Köer

Nedan kommer de olika köernas och deras iteratorers prestanda att analyseras och sedan jämföras med varandra. De jämförs ur tre aspekter. Genomsnittligt antalet iterationer per sekund, genomsnittligt antalet enqueue och dequeue operationer per sekund samt genomsnittligt antal element per iteration.

5.3.1 Framework

Framework kön visade sig vara den kö som helt klart är snabbast på att utföra enqueue och dequeue operationer. Detta går att se i andra grafen inom varje grupp, exempelvis Graf 4.2. Iteratorn för denna kö presterar jämnt och har inte lika stora avvikelser som vissa av de andra köerna som testats. För det mesta så presterade framework kön bättre än den kö som bygger på immutable men sämre än både Scan and Return och Double Collect. Vid initial storlek noll var den kö som bygger på immutable oftast bättre än framework kön. Detta beror antagligen på det faktum att den kö som bygger på immutable vid dessa experiment hade färre element i sig än framework kön. Denna kö var den enda kö som fick bättre resultat vid enqueue och dequeue när trådantalet ökade från 2 till 4. Detta betyder att Framework kön är den enda av de testade köerna där prestandan skalar, dock endast lite. När trådantalet ökar ytterligare så minskar dock prestandan vid de allra flesta parameteruppsättningarna, men fortsätter vara klart bättre än de övriga testade köerna. Framework kön är den kö som för det mesta hade flest element i sig vid iteration, framför allt vid initial storlek på noll. Detta går att se i tredje grafen inom varje grupp, exempelvis Graf 4.3. Detta kan tänkas bero på det faktum att framework kön är den kö som hinner med flest antal enqueue och dequeue operationer under testerna. Vilket kan tänkas leda till att kön då hinner fylla upp och få ett stabilt antal element tidigare i experimenten jämfört med de andra testade köerna.

5.3.2 Kö som bygger på Immutable

Den kö som bygger på immutable visar sig vara långsammast vad gäller enqueue och dequeue operationerna för samtliga parameteruppsättningar. Vid trådantalet 2 är den dock alltid närmare de andra köerna än vid högre trådantal. Iteratorn för denna kö presterar väldigt varierat beroende

på hur många element som finns i kön. I de experiment där initial storlek har varit satt till noll har den varit snabbare än framework kön vid trådantal större än fyra. Den har även presterat bättre än både Scan and Return och Double Collect vid vissa av dessa experiment, exempelvis i Graf 4.10. Vid de övriga initiala storlekarna har den presterat sämst av de testade köerna. Detta kan tänkas bero på att denna kö vid iteration behöver vända en av de stackar den använder för att lagra elementen internt. I värsta fall behöver den alltså gå igenom hela kön två gånger vid en iteration. Det går att se i Graf 4.1 att den kö som bygger på immutable är den kö som faktiskt tappar minst prestanda när trådantalet stiger. Detta var förväntat eftersom denna kös iterator är implementerat på ett sådant sätt att den inte påverkas direkt av arbetstrådarnas operationer.

5.3.3 Scan and Return och Double Collect

Både Scan and Return och Double Collect köerna bygger på Michael och Scott kön (Michael & Scott 1996). På grund av detta finns det många likheter i prestandan mellan dessa två köer, framför allt i antalet enqueue och dequeue operationer respektive kö hinner med. Scan and Return iteratorn är för det allra mesta snabbare än Double Collect iteratorn. Detta är inte förvånande med tanke på att den inte behöver göra kontroller för huruvida kön har förändrats. Double Collect har istället två stycken kontroller den gör där arbetstrådarna kan ställa till det och försämra iteratorns prestanda. I vissa av experimenten har dock Double Collect iteratorn varit snabbare än Scan and Return iteratorn. Detta kan tänkas bero dels på slumpen. Men även på att Scan and Return iteratorn kan bli sovande en längre tid efter att ha hämtat huvudpekaren men innan svanspekaren har hämtats. Eftersom det tillstånd som returneras i detta fall kan innehålla många mer element än vad som faktiskt var i kön vid någon tidpunkt kan iteratorn få sämre prestanda än Double Collect. Iteratorerna i Scan and Return och Double Collect presterar båda två bättre än de övriga två testade köernas iteratorer i de allra flesta experimenten. Framför allt vid höga trådantal presterar dessa två köer bättre än Framework kön och den kö som bygger på immutable. Detta går bland annat att se i Graf 4.13 och Graf 4.22. Scan and Return och Double Collect hinner båda två med ungefär lika många enqueue och dequeue operationer, dock finns det ett fåtal avvikelser. De fall där Double Collect hinner med fler än Scan and Return kan förklaras med att Double Collect iteratorn hjälper svanspekaren hinna ikapp om den halkar efter. Detta gör då att arbetstrådarna inte behöver göra det och kan därför hinna med något fler enqueue och dequeue operationer. De fallen då Scan and Return hinner med fler enqueue och dequeue operationer kan dock inte förklaras på något annat sätt än slumpen. Antalet enqueue och dequeue operationer som Scan and Return och Double Collect köerna hann med ligger över den kö som bygger på immutable men under Framework kön. Prestandan i arbetstrådarna är som bäst vid lågt trådantal och minskar när antalet trådar ökar.

5.3.4 Sammanfattning av köer

Det är intressant att se att samtliga köer förutom framework kön blev konstant sämre på att utföra enqueue och dequeue operationer vid ökat antal trådar. Med andra ord är det bara framework kön som har tillräckligt effektiva enqueue och dequeue operationer för att man skall vinna prestanda på att ha flera trådar som arbetar mot kön. Framework kön var även den kö som vid samtliga experiment hann med flest antal enqueue och dequeue operationer. Vid höga trådantal var den gott och väl över dubbelt så snabb som de övriga testade köerna. Exempelvis i Graf 4.8 utförde Framework kön vid 22 trådar cirka 4,75 miljoner enqueue och lika många dequeue operationer per sekund, medan Scan and Return som gjorde näst mest utförde 1,32 miljoner enqueue och lika många dequeue operationer per sekund. Double Collect ligger väldigt nära Scan and Return med endast fem tusen färre enqueue och dequeue operationer. Den kö som bygger på immutable presterar sämst med 0,39 miljoner enqueue och lika många dequeue operationer per sekund.

Scan and Return hade i nästan alla experiment den snabbaste iteratorn. Double Collect iteratorn presterar för det allra mesta näst snabbast efter Scan and Return. Den kö som bygger på immutable presterade något bättre i några fall med initial storlek noll.

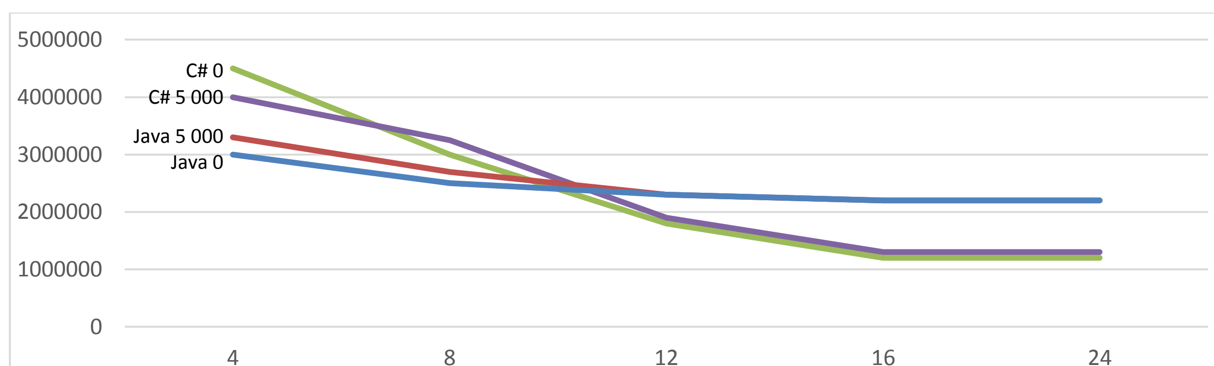
Den kö som bygger på immutable visar sig vara snabbast i väldigt få fall. Antalet enqueue och dequeue operationer den hinner med är i samtliga experiment det lägsta av de testade köerna. I de experimenten med många element i kön visar sig dess iterator vara klart långsammast. Endast i det fall där det är en liten kö presterar denna kös iterator snabbast av de testade köernas iteratorer.

5.4 Jämförelse med Java

All jämförelse som gjorts med resultaten från Nikolakopoulos et al. (2013) har gjorts med pinning strategin none. Detta på grund av att de inte använde några pinning strategier i sina tester, vilket i vårt fall blir samma som pinning strategin none. De initiala storlekarna som finns med i båda studierna är 0 och 5 000 så det är vid dessa jämförelser har gjorts. Nikolakopoulos et al. (2013) testade både med och utan en iteratortråd, något som inte gjordes i vår studie. De testade även med både hög och medelstor belastning, medan vi endast testade med hög belastning. Det är alltså resultaten från deras tester med iteratortråd och hög belastning som jämförts med våra resultat. Då det är Scan and Return och Double Collect som har testats i båda studierna är det dessa köer med iterator vi jämför. Det är även värt att nämna att de kört sina experiment på annan hårdvara än den som använts i våra experiment.

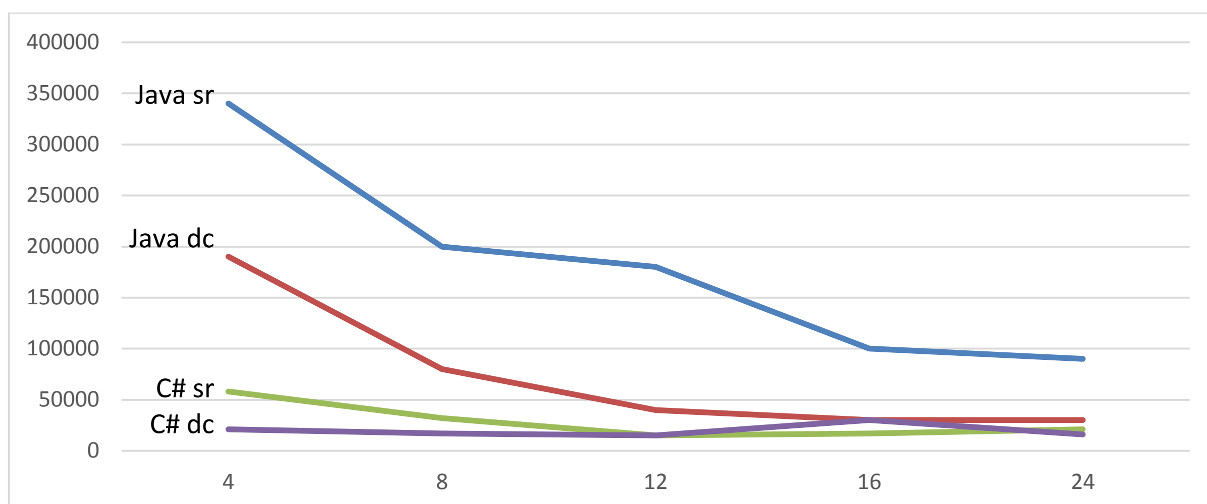
För att visualisera skillnaderna mellan dessa två studier har ett par grafer konstruerats. X-axeln i båda representerar trådantal. Vad Y-axeln representerar beskrivs i bildtexterna.

Vid jämförelse mellan enqueue och dequeue operationernas prestanda i de olika miljöerna framgår det att dessa operationer har en jämnare prestationskurva i Java experimenten. Eftersom Nikolakopoulos et al. (2013) inte skiljt på enqueue och dequeue operationer utan bara presenterar ett medelvärde mellan dessa, har även ett medelvärde från våra resultat fått användas. Då Scan and Return och Double Collect vid denna jämförelse hade relativt lika värden har ett medelvärde använts. Det visar sig att köerna presterar ungefär 25 % snabbare i C# vid låga trådantal men tappar till att vara ungefär 40 % långsammare i C# än i Java vid de högre trådantalen. Dessa jämförelser stämmer för båda initiala storlekarna. Denna jämförelse går att se i Graf 5.1.



Graf 5.1 – Genomsnittligt antal enqueue/dequeue operationer per sekund, Java och C#

Jämförelse mellan iteratorerna kunde endast göras vid initial storlek 0 då 5 000 saknades för iteration i Nikolakopoulos et al. (2013). Det visar sig vid denna jämförelse att iteratorerna var snabbare i Java miljön. Detta går att se i Graf 5.2.

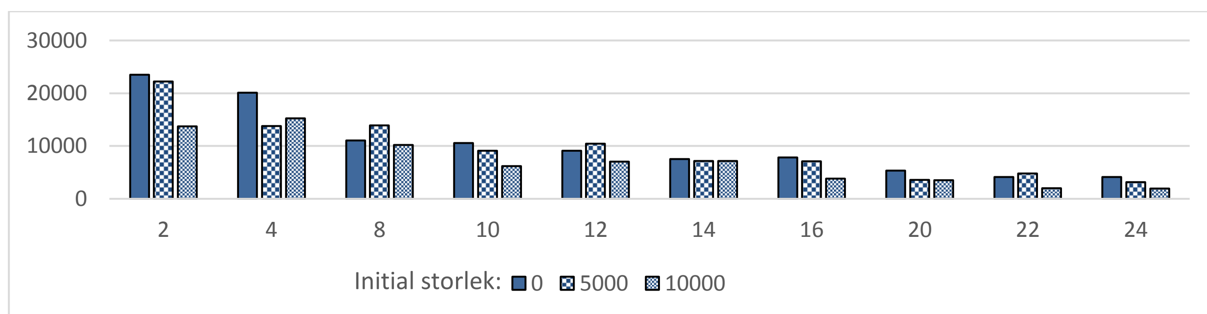


Graf 5.2 – Genomsnittligt antal iterationer per sekund, Java och C#

Prestandan hos enqueue och dequeue operationerna hos dessa köer skiljde sig inte mycket åt mellan de olika miljöerna. Iteratorerna hade dock större skillnad. Framför allt vid låga trådantal presterade iteratorerna i Java snabbare än i C#. Även vid de högre trådantalen stämmer detta men inte i lika stor utsträckning. Då båda experimenten utfördes på datorer utrustade med nehalem processorer borde inte skillnaden i hårdvara orsaka denna skillnad. Istället tror vi författare att orsaken ligger i de olika programspråken. Java verkar vara bättre på att hantera iteration över delad data än C#

5.5 Sammanfattning av analys

Under förutsättningarna att det utförs mest enqueue och dequeue operationer på kön, eller att det är dessa som är viktigast att de är snabba, visar sig framework kön vara det bästa alternativet. Iteratorn hos denna kö är inte den snabbaste men den lämnar garantier för ett atomiskt snapshot. Den skalar även bra med ökad storlek på kön, detta går att se i Graf 5.3.

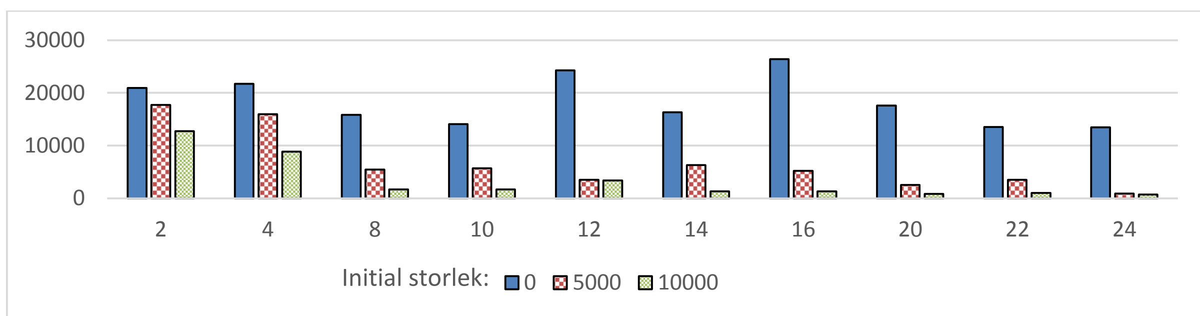


Graf 5.3 – Genomsnittligt antal iterationer per sekund för Framework kön med pinning strategin none

Om iteration istället är det som är viktigast att det går snabbt för så står valet mellan Scan and Return och Double Collect. Scan and Return är den snabbaste iteratorn medan Double Collect för det mesta är den näst snabbaste. Scan and Return lämnar dock ingen garanti för att det tillstånd som returneras är ett atomiskt snapshot av kön. Vad gäller enqueue och dequeue operationer är dessa köer klart långsammare än framework kön, men något snabbare än den kö som bygger på immutable. Mellan dem är det liten skillnad, men denna skillnad är inte signifikant. Double Collect har en liten hjälp för om svansen halkar efter och därför kan denna möjligtvis ha lite snabbare enqueue och dequeue operationer än Scan and Return. Dock blir det

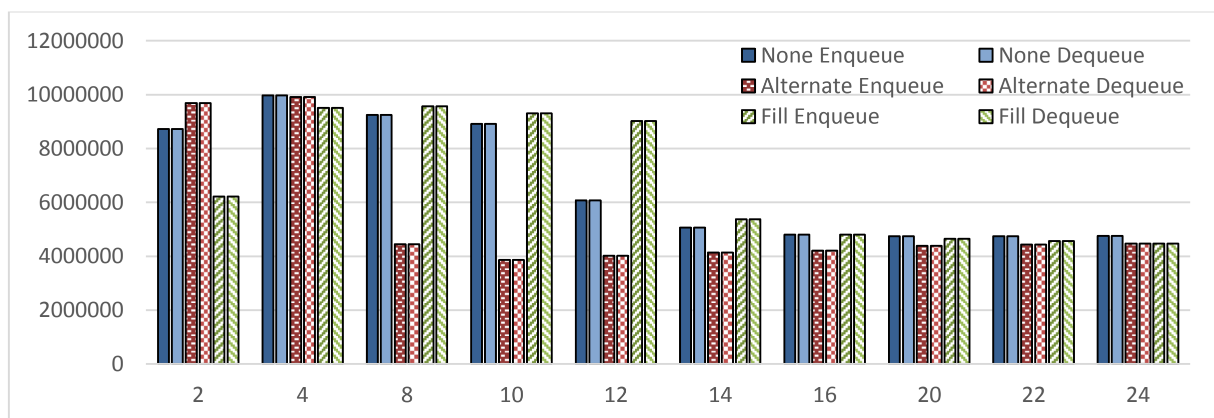
bara snabbare när iteratorn försöker hämta ett tillstånd att iterera. Denna skillnad är för liten för att visa sig i experimenten.

Iteratorn för kön som bygger på immutable visar sig vara bra vid lågt antal element i kön, och endast då. När den initiala storleken på kön ökar minskar iteratorns prestanda markant. Som det går att se i Graf 5.4 är denna försämring närvarande vid samtliga trådantal, men den är tydligare desto högre trådantalet är. Den är ändå för det mesta långsammare än både Scan and Return och Double Collect. Framework kön presterar den bättre än vid initial storlek på noll, men sämre vid övriga initiala storlekar. Antalet enqueue och dequeue operationer som den kö som bygger på immutable hann med var i samtliga fall förutom ett lägst av de testade köerna. Det fall där den inte var långsammast syns i Graf 4.2. Det finns ingen tydlig anledning till att den skulle prestera bättre i just det här experimentet så denna anomali skylls på slumpen. I detta fall var den ändå inte mycket snabbare än den långsammaste, och det var endast en kö som var långsammare. Prestandamässigt är detta alltså inte en kö som vi författare rekommenderar.



Graf 5.4 – Genomsnittligt antal iterationer per sekund för kön som bygger på immutable med pinning strategin none

Då skillnaderna mellan de olika pinning strategierna inte var speciellt stora i de flesta fallen, som går att se i Graf 5.5, går det att konstatera att det är lämpligt att låta Windows sköta pinningen. Dock finns det prestanda att vinna om det är viktigt att programmet är så optimerat som möjligt. Hur mycket prestanda som kan vinnas med hjälp av olika pinning strategier varierar självfallet med vilken hårdvara som används. Därför rekommenderar vi författare att utvecklaren testar sig fram om denne vill optimera sitt program med hjälp av just pinning strategier. För hårdvaran som använts i experimenten hade vi rekommenderat strategin fill, förutsatt att fler än 2 trådar skulle arbeta mot kön. Om endast 2 eller färre trådar skulle användas hade vi rekommenderat strategin none.



Graf 5.5 – Genomsnittligt antal enqueue och dequeue operationer per sekund för Framework kön med initial storlek 5 000

Notera att kön som visualiserats i grafen ovan är framework kön. Denna kö hade vid samtliga parameteruppsättningar en ökning i prestanda mellan 2 och 4 trådar.

De olika initiala storlekarna som använts under experimenten visade sig påverka prestandan för iteratorerna. Desto större initial storlek desto färre antal iterationer hanns med. Detta är en konsekvens av att det blev fler element att iterera över. Vad gäller enqueue och dequeue operationerna har antalet lyckade operationer inte påverkats. Med större initial storlek på kön har antalet misslyckade dequeue operationer minskat, detta beror på att kön mer sällan blir tom.

Hur många arbetstrådar som används påverkar inte de olika iteratorerna på något generellt sätt. Dock påverkas enqueue och dequeue operationerna desto mer. Antalet enqueue och dequeue operationer som hinns med minskar tydligt när antalet trådar ökar. Framework kön han dock med fler enqueue och dequeue operationer vid fyra trådar än vid två trådar, sedan minskar antalet när trådantalet ökar.

6 Slutsats

Vi kan med hjälp av resultaten konstatera att inte någon av de testade köerna är den globalt bästa. De olika köerna har sina styrkor och svagheter vilket gör de olika bra vid olika tillämpningar och situationer. Trots detta så har samtliga frågeställningar som togs fram i början av studien kunnat besvaras.

En av frågeställningarna var *hur stor trade-off är det mellan prestanda och garanti för konsistens?* Kostnaden för att få konsistensgaranti mäter vi i skillnaden mellan Scan and Return och Double Collect iteratorerna. Detta på grund av att dessa är de två snabbaste iteratorerna och Scan and Return inte lämnar garantin medan Double Collect gör det. Denna kostnad visar sig vara relativt stor, Scan and Return presterar upp emot tre gånger så snabbt som Double Collect.

Den andra frågeställningen var *hur förhåller sig prestandan på dessa köer mot varandra?* Denna fråga har svarats mer ingående ovan men det korta svaret är att vid iteration är Scan and Return snabbast, Double Collect är näst snabbast och Framework och den kö som bygger på immutable ligger varierande trea och fyra. Vad gäller enqueue och dequeue operationer är Framework kön är snabbast, Scan and Return och Double Collect ligger delad tvåa och den kö som bygger på immutable är långsammast på dessa operationer.

Den tredje frågeställningen var *hur stor påverkan på prestandan har olika pinning strategier, initiala storlekar samt trådantal?* Det går att utläsa är att alla dessa påverkar prestandan på något sätt. Vilken pinning strategi som används påverkar helt klart resultaten. För det mesta visar sig strategin fill vara bäst medan strategin none är bäst i ett par av fallen. Initial storlek påverkar prestandan så att vid större initial storlek försämras prestandan hos iteratorerna, framför allt för iteratorn hos kön som bygger på immutable. Trådantalet påverkar inte iteratorernas prestanda märkvärt. Dock påverkas enqueue och dequeue operationerna desto mer. Desto fler trådar som används desto färre enqueue och dequeue operationer hinns med.

Med hjälp av svaren på dessa tre frågor kan nu utvecklare göra väl informerade beslut rörande vilken kö med iterator de skall använda i sina system.

6.1 Diskussion

Vid analys av testresultaten visade det sig finnas några anomalier som berodde på slumpen. Alltså kan vi fastslå att experimenten borde kört under längre tid, och fler benchmarks för samma parameteruppsättningar. Detta har vi dock inte kunnat göra i denna studie då det inte funnits tillräckligt med tid för detta.

6.1.1 Reliabilitet

Samtliga experiment i studien har utförts två gånger. Första gången för att samla in datan och andra gången för att verifiera att datan stämmer. Vi kunde i denna verifiering se att värdena kunde skilja sig en del från den första omgången. Dock stämde alltid alla trender och förhållanden överens mellan första och andra omgången. Vi kan alltså säga att de trender och förhållanden mellan köerna som påträffats är reliabla. Trots de ovan nämnda anomalier som uppstod på grund av slumpen går det alltså att se samma trender och förhållanden mellan köerna vid upprepade körningar av experimenten.

6.1.2 Reproducerbarhet

Med ovanstående påstående angående reliabilitet i åtanke kan vi med säkerhet säga att experimenten som utförts går att återskapa och då även få samma resultat igen i form av trender

och förhållanden mellan köerna. Den källkod som inte är bifogad är den kod som hanterar användarinput. Vilket är enkelt att implementera eget och påverkar inte testresultaten.

6.1.3 Generalitet

Eftersom målet med studien var att jämföra prestandan mellan kända algoritmer på en viss plattform har fokus aldrig legat på att få generella resultat. Resultaten som experimenten gett är hårt bundna till den hårdvara som använts under experimenten. Även det att slumpen orsakat anomalier i resultaten gör det svårt att kunna jämföra resultaten från denna studie med resultat från andra studier om inte dessa utförts på liknande hårdvara. Förhållandena som har hittats i studien ser vi dock som generella.

Analysen av pinning strategier visade att det fanns ett starkt band mellan pinning strategiernas prestanda och processorns minneshierarki. Det visade sig vara lönande att lägga trådarna på ett sådant sätt att minnet behövde färdas så kort väg som möjligt. I fallet av den här studien innebar det att det var bäst att vänta med att utnyttja båda processorerna tills första processorn var fylld. Det är antagligen generellt för samtlig hårdvara att försöka undvika att behöva flytta datan i minnet.

Vid undersökning av olika initiala storlekars påverkan på iteratorernas prestanda framstod det att fler element i kön leder till färre iterationer per sekund. Detta resultat är högst generellt för iteratorer eftersom alla iteratorer per definition behöver gå igenom samtliga element i det tillstånd som skall itereras.

Då de kunskaper som den här studien har lett fram till rörande iteratorers beteende och prestanda är specifika för datastrukturen kö, är det inte mycket som är generellt överförbart till andra datastrukturer. Dock kan vi med resultaten från den här studien konstatera att för datastrukturer där det krävs mer än en läsning för att läsa strukturens tillstånd, kommer det att finnas en kostnad för att få konsistensgaranti. Utöver detta finner vi författare inga generella beteenden i resultaten.

6.1.4 Validitet

De värden som är uppmätta med hjälp av benchmarkprogrammet är som tidigare nämnt antalet enqueue och dequeue operationer, antalet iterationer per sekund samt antalet itererade element per iteration. Dessa presenteras sedan med hjälp av medelvärdet. Antalet enqueue och dequeue operationer har använts som mått på hur effektiva dessa operationer är. Detta ser vi som ett bra val på mått. Alternativet man har är att mäta tiden det tar för varje operation att utföras. Detta är dock inte ett bra alternativ då det rör som om väldigt korta tider, så korta att det blir svårt att mäta dessa precist. Antal iterationer per sekund valdes att användas för att det var det mått som vi författare ansåg vara det mått som bäst representerade det vi ville mäta. Det var just tiden att hämta iteratorn som vi ansåg vara det mest intressanta att jämföra. Det alternativ som fanns var att istället mäta antalet itererade element per sekund. Då vi författare anser att tiden att hämta iteratorn inte är lika väl representerat i måttet antalet itererade element per sekund som i måttet antal iterationer per sekund, valde vi att använda antal iterationer per sekund. Antal itererade element per iteration används som ett mått på storleken på kön eftersom vi saknar annat sätt att mäta detta. Då de implementerade köerna inte har något sätt att kolla hur många element kön faktiskt innehåller

6.2 Fortsatta studier

I denna studie har inte minnesanvändandet av de olika köerna studerats. Det vore intressant att även jämföra denna aspekt mellan de olika köerna. Detta kan då tänkas vara en bra utgångspunkt i fortsatta studier.

Det vore även intressant att se fortsatta studier som utför liknande tester på andra datastrukturer. Exempelvis datastrukturen stack skulle gå att testa på liknande sätt. En stack liknar en kö på det sätt att vart man kommer åt datan är begränsad. Samtidigt skiljer de sig åt genom det att kön lägger till element på ena sidan och tar bort från andra, medan stacken lägger till och tar bort element på samma ställe.

Som vi har nämnt tidigare föreslog Nikolakopoulos et al. (2013) fyra stycken iteratorer till Michael och Scott kön (Michael & Scott 1996) men endast två av dessa testades i denna studie. De två iteratorer som inte testades i denna studie är iteratorer som får hjälp av enqueue och dequeue operationerna. Dessa iteratorers prestanda och beteende vore intressant att mäta och analysera i en .NET miljö. Även deras påverkan på enqueue och dequeue operationerna vore intressant att analysera.

Referenser

Anderson, J. H. & Moir, M. (1999). Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems,* 10(12), ss. 1317-1332.

Barnes, G. (1993). A Method for Implementing Lock-free Shared-data Structures. *Proceedings of the 5th International ACM Symposium on Parallel Algorithms and Architectures, ACM,* ss. 261-270.

Cederman, D., Gidenstam, A., Ha, P., Sundell, H., Papatriantafilou, M., Tsigas, P. Lock-free Concurrent Data Structures. *arXiv:1302.2757v1 [cs.DC].* [epubl. före tryckning]

Chann, C., Huang, T. & Chen, C. A Practical Nonblocking Queue Algorithm usinc Compare-and-Swap. *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*., ss. 470-475

Duffy, J. (2008). *Concurrent Programming on Windows.* Boston: Addison-Wesley Professional.

Fuller, S. H. & Millett, L. I. (2011). Computing performance: Game over or next level?. *IEEE Computer Society,* 44(1), ss. 31-38.

Georges, A., Buytaert, D. & Eeckhout, L. (2007). Statistically Rigorous java Performance Evaluation. *ACM Sigplan Notices,* 42(10), ss. 57-76.

Giacomoni, J., Moseley, T. & Vachharajani, M. (2007). FastForward for Efficient Pipeline Parallelism. *16th International Conference on Parallel Architecture and Compilation Techniques.,* ss. 407.

Gidenstam, A., Sundell. H. & Tsigas, P. (2010). Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. *Proceedings of the 14th International Conference on Principles Of Distributed Systems.,* ss. 302-317

Gong, C. & Wing, J. M. (1990). A Library of Concurrent Objects and Their Proofs of Correctness. Teknisk Rapport, Computer Science Department, Carnegie Mellon University. CMU-CS-90-151

Herlihy, M. & Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.,* 12(3), ss. 463-492

Herlihy, M. (1991). Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst., 13(1) ss. 124-149*

Herlihy, M. (1993). A Methodology for Implementing Highly Concurrent Data Objects. *ACM Trans. Program. Lang. Syst.,* 15(5), ss. 745-770

Herlihy, M. & Shavit, N. (2008). *The Art of Multiprocessor Programming.* Burlington: Morgan Kaufmann.

Herman, T. & Damian-Iordache, V. (1997). Space-optimal Wait-free Queues. *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing.,* ss. 280

Hoffman, M., Shalev, O. & Shavit N. (2007). The Baskets Queue. *Proceedings of the 11th International Conference on Principles of Distributed Systems*, ss. 401-414

Intel, (u.å.). *concurrent_queue Template Class*. http://www.threadingbuildingblocks.org/docs/help/reference/containers_overview/concurrent_queue_cls.htm [2014-05-15]

Lamport, L. (1983). Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.,* 5(2), ss. 190-222.

Lippert, E. (2007). Immutability in C# Part Four: An Immutable Queue. *Eric Lippert's Blog* [blogg], 10 december. http://blogs.msdn.com/b/ericlippert/archive/2007/12/10/immutability-in-c-part-four-an-immutable-queue.aspx [2014-04-29]

McGachey, P. & Hosking, A. L. (2006). *Reducing Generational Copy Reserve Overhead with Fallback Compaction.* Ottawa, Ontario, Canada, Proceedings of the 5th International Symposium on Memory Management, ACM.

Michael, M. M. (2004). Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallell and Distributed Systems,* 15(6), ss. 491-504.

Michael, M. M. & Scott, M. L. (1996). Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. *Philadelphia, Pennsylvania, USA, Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (PODC '96)*, ss. 267-275.

Microsoft, (u.å. a). *MSDN – ConcurrentQueue(T) Class*. http://msdn.microsoft.com/en-us/library/dd267265(v=vs.100).aspx [2014-04-01]

Microsoft, (u.å. b). *MSDN – ConcurrentQueue(T).GetEnumerator Method*. http://msdn.microsoft.com/en-us/library/dd287144(v=vs.100).aspx [2014-04-01]

Microsoft, (u.å. c). *MSDN - Immutable Collections.* http://msdn.microsoft.com/en-us/library/dn385366(v=vs.110).aspx [2014-04-08]

Microsoft, (u.å. d). *MSDN - ImmutableQueue(T) Class.* Available at: http://msdn.microsoft.com/en-us/library/dn467186(v=vs.110).aspx [2014-04-08]

Microsoft, (u.å. e). *MSDN - Stopwatch Class.* http://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch.aspx [2014-04-11]

Microsoft, (u.å. f). *MSDN - ProcessThread.ProcessorAffinity Property.* http://msdn.microsoft.com/en-us/library/system.diagnostics.processthread.processoraffinity.aspx [2014-04-16]

Moir, M., Nussbaum, D., Shalev, O. & Shavit, N. (2005). Using elimination to implement scalable and lock-free FIFO queues. *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ss. 253-262

Nikolakopoulos, Y., Gidenstam, A., Papatriantafilou, M. & Tsigas, P. (2013). Enhancing Concurrent Data Structures with Concurrent Iteration Operations: Consistency and Algorithms. Teknisk Rapport, Chalmers. 2013:06

Oracle, (u.å.). *ConcurrentLinkedQueue (Java Platform SE 7).* http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html [2014-05-15]

Prakash, S., Lee, Y.H. & Johnson, T. A nonblocking algorithm for shared queues using compare-and-swap., *IEEE Transactions on Computers*, 43(5), ss. 548-559

Thomadakis, M. E. (2011). The architecture of the Nehalem processor and Nehalem-EP smp platforms. *Resource*, 3(2).

Valois, J. D. (1994) Implementing Lock-Free Queues., *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems.,* ss. 64-69

Valois, J. D. (1996) Lock-Free Data Structures. Doktorsavhandling. Rensselaer Polytechnic Institute, Troy, New York.

Högskolan i Borås är en modern högskola mitt i city. Vi bedriver utbildningar inom ekonomi och informatik, biblioteks- och informationsvetenskap, mode och textil, beteendevetenskap och lärarutbildning, teknik samt vårdvetenskap.

På **institutionen Handels- och IT-högskolan (HIT)** har vi tagit fasta på studenternas framtida behov. Därför har vi skapat utbildningar där anställningsbarhet är ett nyckelord. Ämnesintegration, helhet och sammanhang är andra viktiga begrepp. På institutionen råder en närhet, såväl mellan studenter och lärare som mellan företag och utbildning.

Våra **ekonomiutbildningar** ger studenterna möjlighet att lära sig mer om olika företag och förvaltningar och hur styrning och organisering av dessa verksamheter sker. De får även lära sig om samhällsutveckling och om organisationers anpassning till omvärlden. De får möjlighet att förbättra sin förmåga att analysera, utveckla och styra verksamheter, oavsett om de vill ägna sig åt revision, administration eller marknadsföring. Bland våra **IT-utbildningar** finns alltid något för dem som vill designa framtidens IT-baserade kommunikationslösningar, som vill analysera behov av och krav på organisationers information för att designa deras innehållsstrukturer, bedriva integrerad IT- och affärsutveckling, utveckla sin förmåga att analysera och designa verksamheter eller inrikta sig mot programmering och utveckling för god IT-användning i företag och organisationer.

Forskningsverksamheten vid institutionen är såväl professions- som design- och utvecklingsinriktad. Den övergripande forskningsprofilen för institutionen är handels- och tjänsteutveckling i vilken kunskaper och kompetenser inom såväl informatik som företagsekonomi utgör viktiga grundstenar. Forskningen är välrenommerad och fokuserar på inriktningarna affärsdesign och Co-design. Forskningen är också professionsorienterad, vilket bland annat tar sig uttryck i att forskningen i många fall bedrivs på aktionsforskningsbaserade grunder med företag och offentliga organisationer på lokal, nationell och internationell arena. Forskningens design och professionsinriktning manifesteras också i InnovationLab, som är institutionens och Högskolans enhet för forskningsstödjande systemutveckling.

HÖGSKOLAN I BORÅS

VETENSKAP FÖR PROFESSION

BESÖKSADRESS: JÄRNVÄGSGATAN 5 · POSTADRESS: ALLÉGATAN 1, 501 90 BORÅS
TFN: 033-435 40 00 · E-POST: INST.HIT@HB.SE · WEBB: WWW.HB.SE/HIT

Bilagor

Bilaga A – Gloslista

Ord	Beskrivning
Garbage collector	En modul i programspråket som frigör minne som inte längre används.
Garbage collection	En körning av garbage collectorn.
Benchmarkprogrammet	Programmet som mäter prestandan hos de olika köerna, det vill säga det programmet som startar 10 (plus 1) körningar av en parameteruppsättning.
Benchmark	En av de 10 (plus 1) körningarna i benchmarkprogrammet
CAS(destination, komparand, utbyte)	Compare and Swap, en atomisk operation som jämför värdet i destinationen med komparanden och skriver utbytet till destinationen om jämförelsen stämmer.

Bilaga B – Kod

Framework

```
/// <author>
/// Viktor Lodin
/// Magnus Olovsson
/// </author>
/// <summary>
/// The standard implementation of a ConcurrentQueue in Microsoft.NET
/// </summary>
/// <typeparam name="T"></typeparam>
class FrameWorkQueue<T> : IParallelQueue<T>
{
    ConcurrentQueue<T> _myQueue;

    public FrameWorkQueue()
    {
        _myQueue = new ConcurrentQueue<T>();
    }

    public bool Enqueue(T item)
    {
        _myQueue.Enqueue(item);
        return true;
    }

    public bool TryDequeue(ref T item)
    {
        return _myQueue.TryDequeue(out item);
    }

    public IParallelQueue<T> CreateNew()
    {
        return new FrameWorkQueue<T>();
    }

    public IEnumerator<T> GetEnumerator()
    {
        return _myQueue.GetEnumerator();
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return _myQueue.GetEnumerator();
    }
}
```

Immutable

```
/// <author>
/// Viktor Lodin
/// Magnus Olovsson
/// </author>
class ConcurrentImmutableQueue<T> : IParallelQueue<T>
{
    private volatile ImmutableQueue<T> _currentState;

    public ConcurrentImmutableQueue()
    {
        _currentState = ImmutableQueue<T>.Empty;
    }

    public bool Enqueue(T item)
    {
        while (true)
        {
            ImmutableQueue<T> oldState = _currentState;
            ImmutableQueue<T> newState = oldState.Enqueue(item);
            if (Interlocked.CompareExchange(ref _currentState, newState,
                                            oldState).Equals(oldState))
                return true;
        }
    }

    public bool TryDequeue(ref T item)
    {
        while (true)
        {
            ImmutableQueue<T> oldState = _currentState;

            if (oldState.IsEmpty)
                return false;

            ImmutableQueue<T> newState = oldState.Dequeue(out item);
            if (Interlocked.CompareExchange(ref _currentState, newState,
                                            oldState).Equals(oldState))
                return true;
        }
    }

    public IParallelQueue<T> CreateNew()
    {
        return new ConcurrentImmutableQueue<T>();
    }

    public IEnumerator<T> GetEnumerator()
    {
        ImmutableQueue<T> enumeratedState = _currentState;
        foreach (T item in enumeratedState)
        {
            yield return item;
        }
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        ImmutableQueue<T> enumeratedState = _currentState;
        foreach (T item in enumeratedState)
        {
            yield return item;
        }
    }
}
```

Michael och Scott

```
/// <author>
/// Viktor Lodin
/// Magnus Olovsson
/// </author>
abstract class MichaelScottQueue<T> : IParallelQueue<T>
{
    protected class Node
    {
        public T value;
        public volatile Node next;
    }

    protected volatile Node _head;
    protected volatile Node _tail;

    public MichaelScottQueue()
    {
        _head = _tail = new Node();
    }

    public bool Enqueue(T item)
    {
        // Creates a new node for the item to insert
        Node node = new Node()
        {
            value = item,
            next = null
        };

        Node tail, next;

        while (true)
        {
            tail = _tail;
            next = tail.next;

            // Make sure the queue has not changed
            if (tail == _tail)
            {
                // Make sure tail points to the last element of the queue
                if (next == null)
                {
                    // Try to enqueue the new value
                    if (CAS(ref tail.next, next, node))
                    {
                        break;
                    }
                }
                // If the tail is falling behind, this will help it catch up
                else
                {
                    CAS(ref _tail, tail, next);
                }
            }
        }
        // Point tail to the newly enqueued node
        CAS(ref _tail, tail, node);

        return true;
    }
```

```
    public bool TryDequeue(ref T item)
    {
        Node head, tail, next;

        while (true)
        {
            head = _head;
            tail = _tail;
            next = head.next;

            // Make sure the queue has not changed
            if (head == _head)
            {
                if (head == tail)
                {
                    // If head and tail points to the same node and
                    // next is null, the queue is empty
                    if (next == null)
                    {
                        return false;
                    }
                    // If next is not null, tail is falling behind
                    // So we help it catch up
                    CAS(ref _tail, tail, next);
                }
                // We can dequeue the value
                else
                {
                    item = next.value;

                    if (CAS(ref _head, head, next))
                    {
                        return true;
                    }
                }
            }
        }
    }

    public abstract IParallelQueue<T> CreateNew();

    public abstract IEnumerator<T> GetEnumerator();

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    /// <summary>
    /// Performs an atomic Compare and Swap operation
    /// </summary>
    /// <param name="destination"> The address to modify </param>
    /// <param name="comparand"> The value to compare with </param>
    /// <param name="exchange"> The value to write if destination matches comparand
    /// </param>
    /// <returns></returns>
    protected bool CAS(ref Node destination, Node comparand, Node exchange)
    {
        return Interlocked.CompareExchange(ref destination, exchange, comparand)
               == comparand;
    }
}
```

Scan and Return

```
/// <author>
/// Viktor Lodin
/// Magnus Olovsson
/// </author>
/// <summary>
/// The scan and return extension used in the benchmark.
/// This iterator does not leave any guarantee that the
/// returned state is a point in time snapshot of the queue
/// </summary>
class ScanReturnExtension<T> : MichaelScottQueue<T>
{
    public override IEnumerator<T> GetEnumerator()
    {
        Node currentHead;
        Node currentTail;

        currentHead = _head;
        currentTail = _tail;

        // Check if queue is empty
        if (currentHead == currentTail)
            yield break;

        Node next = currentHead;

        do
        {
            next = next.next;
            yield return next.value;
        }
        while (next != currentTail);
    }

    public override MicroBenchmark.IParallelQueue<T> CreateNew()
    {
        return new ScanReturnExtension<T>();
    }
}
```

Double Collect

```
/// <author>
/// Viktor Lodin
/// Magnus Olovsson
/// </author>
/// <summary>
/// The Double collect extension used in the benchmark.
/// This iterator leaves a guarantee that the
/// returned state is a point in time snapshot of the queue.
/// </summary>
class DoubleCollectExtension<T> : MichaelScottQueue<T>
{

    public override IEnumerator<T> GetEnumerator()
    {
        Node currentHead;
        Node currentTail;
        Node tailNext;

        while (true)
        {
            currentHead = _head;
            currentTail = _tail;
            tailNext = currentTail.next;

            // Make sure tail points to the last element of the queue
            if (tailNext == null)
            {
                // Make sure the queue has not changed
                // This guarantee a point in time snapshot
                if (currentHead == _head)
                {
                    break;
                }
            }
            // If the tail pointer is lagging behind, help it catch up
            else
            {
                CAS(ref _tail, currentTail, tailNext);
            }
        }

        // Check if queue is empty
        if (currentHead == currentTail)
            yield break;

        Node next = currentHead;

        // Iterate the queue and return the items
        do
        {
            next = next.next;
            yield return next.value;
        }
        while (next != currentTail);
    }

    public override MicroBenchmark.IParallelQueue<T> CreateNew()
    {
        return new DoubleCollectExtension<T>();
    }
}
```

Benchmark

```
/// <author>
/// Viktor Lodin
/// Magnus Olovsson
/// </author>
public class IteratorBenchmark
{
    private Thread[] _workerThreads;
    private Thread _iterationThread;

    private DateTime _created = DateTime.Now;
    private string _resultsDirectory;
    private bool _reportResults;
    private string _resultsFile;

    /// <summary>
    /// This variable will indicate whether or not
        the benchmark should be running.
    /// All operations on it should be done in a thread-safe manner.
    /// </summary>
    private volatile int _benchmarkIsRunning = 0;

    /// <summary>
    /// Used to tell if all the threads have finished
    /// </summary>
    private int _finishedThreads = 0;

    /// <summary>
    /// Used to tell what number the current benchmark run has
    /// </summary>
    private int _numBenchmark = 0;

    /// <summary>
    /// Used to report how many successful enqueues and dequeues we manage to
    /// do or not during the benchmark
    /// </summary>
    private int _successfullEnqueues = 0;
    private int _successfullDequeues = 0;
    private int _unSuccessfullEnqueues = 0;
    private int _unSuccessfullDequeues = 0;

    /// <summary>
    /// Used to report the min, average and max number of elements that was
    /// iterated through by the iteration thread
    /// </summary>
    private double _minElementsIterated = Double.MaxValue,
                   _avgElementsIterated = 0,
                   _maxElementsIterated = Double.MinValue,
                   _finalElementsIterated = 0;

    /// <summary>
    /// Used to report how many gaps were found during iteration
    /// </summary>
    private int _foundGaps = 0;

    /// <summary>
    /// Used to give the elements a number
    /// </summary>
    private int _elementNumber;
```

```
/// <summary>
/// Used to report how many iterations was made during the benchmark
/// </summary>
private int _numIterations = 0;

/// <summary>
/// The first element in the tuple represents the threads ID,
    if this number is -1 the element was in the queue initially.
/// The second element in the tuple represents the elements number.
/// </summary>
public IParallelQueue<Tuple<int, int>> Queue { get; set; }

/// <summary>
/// The number of worker threads that should be running during the benchmark
/// </summary>
public int NumWorkerThreads { get; set; }

/// <summary>
/// The number of objects that will be in the queue at the beginning
    of every testrun
/// </summary>
public int InitialQueueSize { get; set; }

/// <summary>
/// The number of times the benchmark should be run
/// </summary>
public int NumBenchmarks { get; set; }

/// <summary>
/// The length of the benchmark in milliseconds. The benchmark will
    not force any operation to stop, it will wait for them to finish
/// </summary>
public int LengthOfBenchmarkInMilliseconds { get; set; }

/// <summary>
/// Wheter or not to print a message to the console after each benchmark run
/// </summary>
public bool ReportEachBenchmarkToConsole { get; set; }

/// <summary>
/// What settings to use for pinning threads
/// </summary>
public PinningSettings PinningSettings { get; set; }

public IteratorBenchmark(string resultsDir)
{
    NumWorkerThreads = 3;
    NumBenchmarks = 10;
    ReportEachBenchmarkToConsole = false;

    DirectoryInfo dir = new DirectoryInfo(resultsDir);
    if (!dir.Exists)
        dir.Create();
    _resultsDirectory = resultsDir;
}
```

```
/// <summary>
/// Starts a batch of benchmarks using the current settings
/// </summary>
public void Start()
{
    if (Queue == null)
        throw new InvalidOperationException(
            "Must assign a queue to test before starting the benchmark!");

    Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;

    _workerThreads = new Thread[NumWorkerThreads];

    // Write headers to the result file
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append(_resultsDirectory);
    stringBuilder.Append(_created.ToString("yyyy-MM-dd_HH.mm"));
    stringBuilder.Append(@"_");
    stringBuilder.Append(NumWorkerThreads + 1);
    stringBuilder.Append(@"threads");
    stringBuilder.Append(@".txt");

    _resultsFile = stringBuilder.ToString();

    StreamWriter outFile = new StreamWriter(_resultsFile, false);

    outFile.WriteLine("Parameters set to:");
    outFile.WriteLine("Queue implementation: " + Queue.GetType().ToString());
    outFile.WriteLine("Number of worker threads: " + NumWorkerThreads);
    outFile.WriteLine("Number of benchmark runs: " + NumBenchmarks);
    outFile.WriteLine("Length of each benchmark: " +
                      LengthOfBenchmarkInMilliseconds + "ms");
    outFile.WriteLine("Initial size of the queue: " + InitialQueueSize +
                      " elements");
    outFile.WriteLine("Pinning setings: " + PinningSettings.ToString());
    outFile.WriteLine();

    outFile.WriteLine("Benchmark No.\t" +
                      "Running time\t" +
                      "Successfull Enqueues\t" +
                      "Unsuccessfull Enqueus\t" +
                      "Successfull Dequeues\t" +
                      "Unsuccessfull Dequeus\t" +
                      "No. of iterations\t" +
                      "No. of gaps found\t" +
                      "Min No. of elements\t" +
                      "Avg No. of elements\t" +
                      "Max No. of elements\t" +
                      "Final No. of elements\t");

    outFile.Close();

    // We want to do one benchmark first without reporting the results
    // This is because we dont want the JIT compilation to be a part of
       the results.
    _reportResults = false;
```

```
    for (_numBenchmark = 0; _numBenchmark < NumBenchmarks + 1;
         Interlocked.Increment(ref _numBenchmark))
    {
        // reinitialize the queue
        Queue = Queue.CreateNew();

        // fill queue with initial data
        for (_elementNumber = 0; _elementNumber < InitialQueueSize;
             Interlocked.Increment(ref _elementNumber))
        {
            Queue.Enqueue(new Tuple<int, int>(0, _elementNumber));
        }

        GC.Collect();
        GC.WaitForPendingFinalizers();
        GC.Collect();

        DoWork();

        if (ReportEachBenchmarkToConsole)
        {
            Console.WriteLine("Benchmark #" + _numBenchmark + " finished!");
        }

        _reportResults = true;
    }
}

/// <summary>
/// This method performs one benchmark
/// </summary>
private void DoWork()
{
    for (int i = 0; i < _workerThreads.Length; i++)
    {
        _workerThreads[i] = new Thread(
            new ParameterizedThreadStart(WorkerMethod));
        _workerThreads[i].Priority = ThreadPriority.AboveNormal;
    }

    _iterationThread = new Thread(new ThreadStart(IterationMethod));
    _iterationThread.Priority = ThreadPriority.AboveNormal;

    _finishedThreads = 0;
    _successfullEnqueues = 0;
    _successfullDequeues = 0;
    _unSuccessfullEnqueues = 0;
    _unSuccessfullDequeues = 0;
    _minElementsIterated = Double.MaxValue;
    _avgElementsIterated = 0;
    _maxElementsIterated = Double.MinValue;
    _foundGaps = 0;

    for (int i = 0; i < _workerThreads.Length; i++)
    {
        _workerThreads[i].Start(i + 1);
    }

    _iterationThread.Start();
```

```
        Stopwatch sw = Stopwatch.StartNew();

        // signal to all threads to start their work
        Interlocked.Exchange(ref _benchmarkIsRunning, 1);

        // let this thread sleep for the length of the benchmark
        Thread.Sleep(LengthOfBenchmarkInMilliseconds);

        // signal to all threads to stop their work
        Interlocked.Exchange(ref _benchmarkIsRunning, 0);

        // wait for all threads to finish their current work
        while (!Interlocked.Equals(_finishedThreads, NumWorkerThreads + 1)) { }

        sw.Stop();

        // wait for worker threads to report their results
        for (int i = 0; i < NumWorkerThreads; i++)
        {
            _workerThreads[i].Join();
        }

        // wait for the iteration thread to report its results
        _iterationThread.Join();

        if (_reportResults)
        {
            StreamWriter outFile = new StreamWriter(_resultsFile, true);

            const char tab = '\t';

            outFile.Write("" + _numBenchmark + tab);
            outFile.Write("" + sw.ElapsedMilliseconds + tab);
            outFile.Write("" + _successfullEnqueues + tab);
            outFile.Write("" + _unSuccessfullEnqueues + tab);
            outFile.Write("" + _successfullDequeues + tab);
            outFile.Write("" + _unSuccessfullDequeues + tab);
            outFile.Write("" + _numIterations + tab);
            outFile.Write("" + _foundGaps + tab);
            outFile.Write("" + _minElementsIterated + tab);
            outFile.Write("" + _avgElementsIterated + tab);
            outFile.Write("" + _maxElementsIterated + tab);
            outFile.WriteLine("" + _finalElementsIterated);

            outFile.Close();
        }

    }
```

```
/// <summary>
/// The method run by all the worker threads.
/// Performs enqueues and dequeues on the queue with equal probability.
/// </summary>
/// <param name="threadID">
/// The id of the thread, this is used to determine what core
/// to run on if using a pinning strategy other then none.
/// </param>
private void WorkerMethod(object threadID)
{
    int myThreadID = (int)threadID;
    int seqNo = 0, numEnqueues = 0;
    int successfulDequeus = 0, numDequeues = 0;
    Thread.CurrentThread.Name = "Benchmark #" + _numBenchmark +
                                " Workerthread #" + myThreadID;

    // set processor affinity if using pinning strategy other then none
    const int numCores = 12;
    if (PinningSettings == PinningSettings.Fill)
    {
        SetProcessorAffinity(myThreadID);
    }
    else if (PinningSettings == PinningSettings.Alternate)
    {
        if (myThreadID < numCores)
            SetProcessorAffinity(2 * myThreadID);
        else
            SetProcessorAffinity((myThreadID - numCores) * 2 + 1);
    }

    Random random = new Random(_created.Millisecond +
        (_numBenchmark * NumWorkerThreads) + myThreadID);
    while (_benchmarkIsRunning == 0) { }

    // Run until stopped
    Tuple<int, int> returnItem = new Tuple<int, int>(0, 0);
    while (_benchmarkIsRunning == 1)
    {
        if (random.NextDouble() >= 0.5)
        {
            numDequeues++;
            if (Queue.TryDequeue(ref returnItem))
            {
                successfulDequeus++;
            }
        }
        else
        {
            numEnqueues++;
            if (Queue.Enqueue(new Tuple<int, int>(myThreadID, seqNo + 1)))
            {
                seqNo++;
            }
        }
    }
```

```
        // announce that the current thread is finished
        Interlocked.Increment(ref _finishedThreads);

        // report the results
        Interlocked.Add(ref _successfullEnqueues, seqNo);
        Interlocked.Add(ref _unSuccessfullEnqueues, numEnqueues - seqNo);

        Interlocked.Add(ref _successfullDequeues, successfulDequeus);
        Interlocked.Add(ref _unSuccessfullDequeues,
                        numDequeues - successfulDequeus);
    }

    /// <summary>
    /// The method run by the iteration thread.
    /// It iterates the queue constantly
    /// </summary>
    private void IterationMethod()
    {
        Thread.CurrentThread.Name = "Benchmark #" + _numBenchmark +
                                    " iteration thread";
        if (PinningSettings != PinningSettings.None)
            SetProcessorAffinity(0);

        int[] lastFoundSeqNo = new int[NumWorkerThreads + 1];
        int diff;

        while (_benchmarkIsRunning == 0) { }

        _numIterations = 0;
        uint itemsInIterations;
        List<uint> resultsItems = new List<uint>();
        while (_benchmarkIsRunning == 1)
        {
            _numIterations++;
            itemsInIterations = 0;

            for (int i = 0; i < NumWorkerThreads + 1; i++)
            {
                lastFoundSeqNo[i] = 0;
            }

            foreach (Tuple<int, int> item in Queue)
            {
                if (lastFoundSeqNo[item.Item1] != 0)
                {
                    diff = item.Item2 - lastFoundSeqNo[item.Item1];
                    if (diff != 1)
                        _foundGaps += diff - 1;
                }
                lastFoundSeqNo[item.Item1] = item.Item2;

                itemsInIterations++;
            }
            resultsItems.Add(itemsInIterations);
        }

        // announce that the current thread is finished
        Interlocked.Increment(ref _finishedThreads);
```

```
        // report the results
        if (_reportResults)
        {
            const int partSize = 1000;

            int m;
            double partialSum, sumOfParts = 0, numberOfParts = 0;
            for (int i = 0; i < _numIterations; i += partSize)
            {
                partialSum = 0;
                for (m = 0; m < partSize; m++)
                {
                    if (i + m >= _numIterations)
                        break;

                    if (resultsItems.ElementAt(i + m) > _maxElementsIterated)
                        _maxElementsIterated = resultsItems.ElementAt(i + m);
                    else if (resultsItems.ElementAt(i + m) < _minElementsIterated)
                        _minElementsIterated = resultsItems.ElementAt(i + m);

                    partialSum += resultsItems.ElementAt(i + m);
                }

                sumOfParts += (partialSum / (double)m) * ((double)m / partSize);
                numberOfParts += ((double)m / partSize);
            }

            _avgElementsIterated = sumOfParts / numberOfParts;
            _finalElementsIterated = resultsItems.ElementAt(_numIterations - 1);

        }
    }

    /// <summary>
    /// A helper method to set the processor affinity of the current thread
    /// </summary>
    /// <param name="processorIndex">
    /// The index of the processor we want the current thread to run on
    /// </param>
    private void SetProcessorAffinity(int processorIndex)
    {
        int myThreadID = AppDomain.GetCurrentThreadId();
        // This method is depricated, but the alternative Microsofts
        // recommends doesn't yield the correct result
        // Microsoft recommends using Thread.CurrentThread.ManagedThreadId;

        ProcessThreadCollection threads = Process.GetCurrentProcess().Threads;
        foreach (ProcessThread processThread in threads)
        {
            if (processThread.Id == myThreadID)
            {
                processThread.ProcessorAffinity = new IntPtr(
                    (int)Math.Pow(2, processorIndex));
                return;
            }
        }
    }
}
```

Gränssnitt kö

```
/// <author>
/// Viktor Lodin
/// Magnus Olovsson
/// </author>
public interface IParallelQueue<T> : IEnumerable<T>
{
    /// <summary>
    /// Enqueues item into the queue
    /// </summary>
    /// <param name="item"></param>
    /// <returns>
    /// returns true or false indicating whether or not the
    /// operation succeeded
    /// </returns>
    bool Enqueue(T item);

    /// <summary>
    /// Dequeses the first item in the queue and returns it.
    /// </summary>
    /// <param name="item">
    /// The dequeued item, null if the queue was empty
    /// </param>
    /// <returns>
    /// returns true or false indicating whether or not the
    /// operation succeeded
    /// </returns>
    bool TryDequeue(ref T item);

    /// <summary>
    /// Creates and returns a new instance of the queue
    /// </summary>
    /// <returns></returns>
    IParallelQueue<T> CreateNew();
}
```