

Тема вежбе: Тutorials – Интел алати за паралелизацију програма

## ОТКРИВАЊЕ ПОГОДНИХ МЕСТА У ПРОГРАМСКОМ КОДУ ЗА ПАРАЛЕЛНУ ОБРАДУ И ЊЕНА ПРИМЕНА

### Проналажење погодних места за паралелну обраду

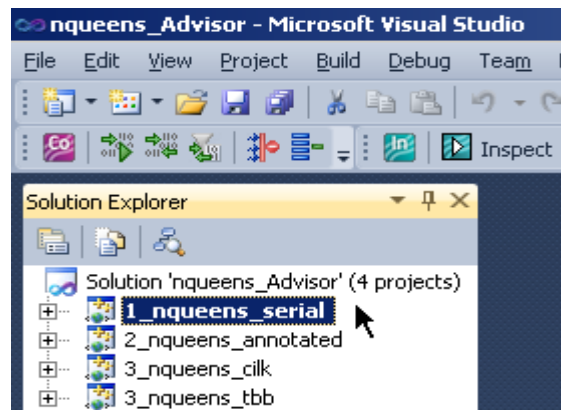
Да би смо пронашли делове програмског кода који троши значајнији део времена извршења програма, користимо алат за мерење активности програма (тзв. *Survey tool*).

Ова фаза се састоји од два корака:

1. Отварање и превожње пројекта
2. Покретање чаробњака *Advisor Workflow*

#### Отварање и превожње пројекта са *Survey* алатом:

1. Отворите пројекат помоћу команде File > Open > Project/Solution:
2. Отворите решење *nqueens\_Advisor.sln*
3. Кликните десним дугметом на пројекат *1\_nqueens\_serial* из овог решења, као на слици 1.



Слика 1: Избор пројекта


4. Из контекстног менија одаберите опцију *Set as Startup Project*
5. Из главног менија *Build* одаберите *Configuration Manager*
6. Затим одаберите *Release* конфигурацију за овај пројекат

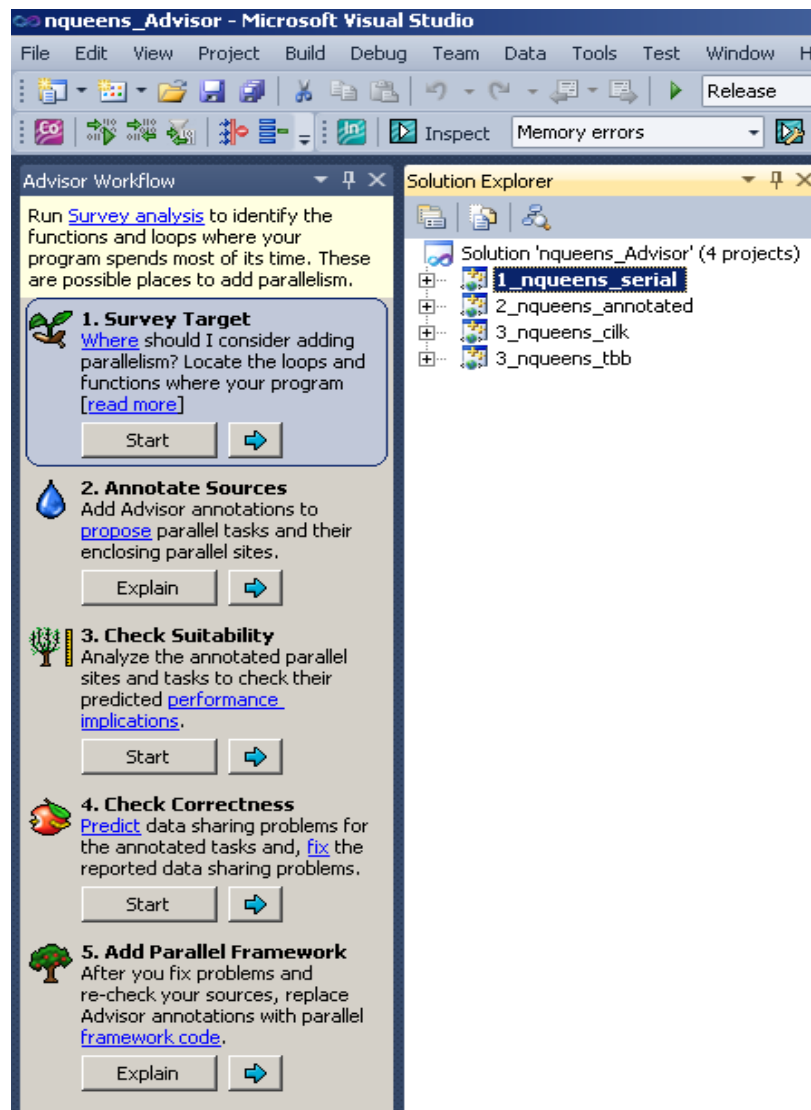
**НАПОМЕНА:** Уколико користите *Debug* конфигурацију са овим алатом, можете добити резултате који не осликавају право стање.

7. Из главног менија **Build** одаберите **Build 1\_nqueens\_serial**
8. Проверите садржај прозора **Output** да ли има грешака у превођењу
9. На крају, програм извршити одабиром команде **Debug > Start Without Debugging**

### Покретање чаробњака *Advisor Workflow*

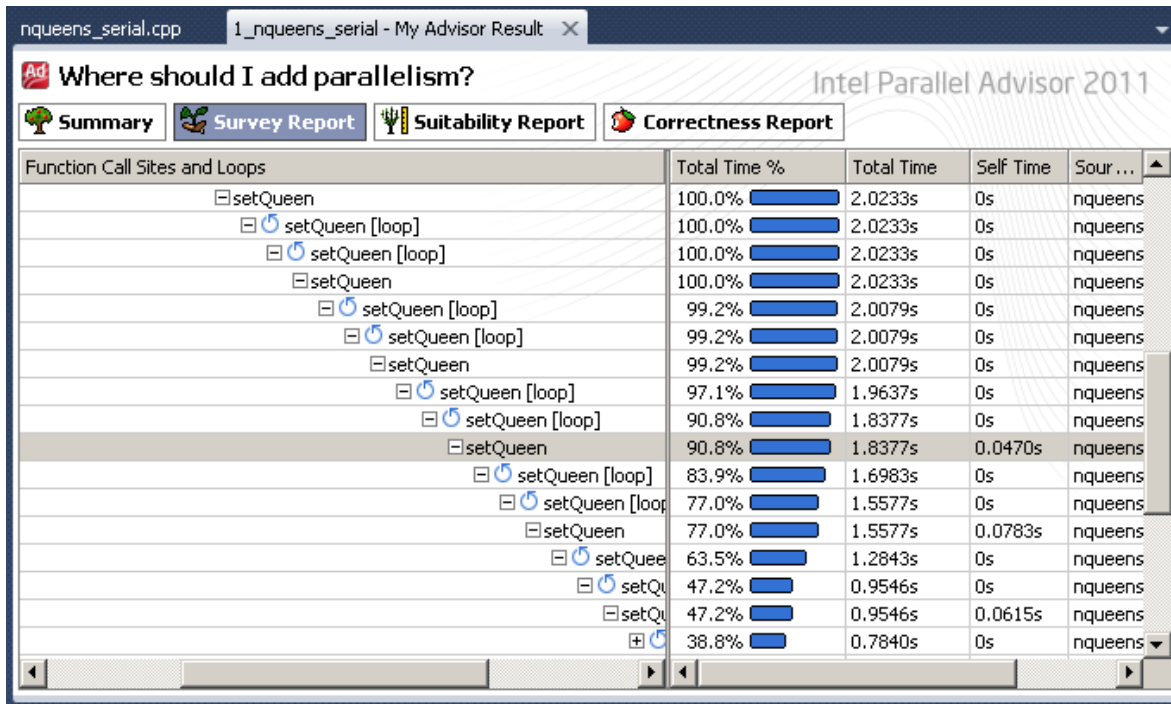
Чаробњак (слика 2) се може покренути на два начина:

- Притиском левог тастера миша на иконицу , или
- Одабиром команде **Tools > Parallel Advisor 2011 > Open Advisor Workflow**



Слика 2: Изглед чаробњака *Advisor Workflow*

Анализу програмског кода покрећемо притиском на дугме **Start**. Анализатор ће покренути програм како би прикупио податке о трајању извршавања различитих делова кода. Подаци се приказују у оквиру извештаја рада (**Survey Report - My Advisor Result**).



Function Call Sites and Loops	Total Time %	Total Time	Self Time	Source
setQueen	100.0%	2.0233s	0s	nqueens
setQueen [loop]	100.0%	2.0233s	0s	nqueens
setQueen [loop]	100.0%	2.0233s	0s	nqueens
setQueen	100.0%	2.0233s	0s	nqueens
setQueen [loop]	99.2%	2.0079s	0s	nqueens
setQueen [loop]	99.2%	2.0079s	0s	nqueens
setQueen	99.2%	2.0079s	0s	nqueens
setQueen [loop]	97.1%	1.9637s	0s	nqueens
setQueen [loop]	90.8%	1.8377s	0s	nqueens
setQueen	90.8%	1.8377s	0.0470s	nqueens
setQueen [loop]	83.9%	1.6983s	0s	nqueens
setQueen [loop]	77.0%	1.5577s	0s	nqueens
setQueen	77.0%	1.5577s	0.0783s	nqueens
setQueen	63.5%	1.2843s	0s	nqueens
setQueen	47.2%	0.9546s	0s	nqueens
setQueen	47.2%	0.9546s	0.0615s	nqueens
setQueen	38.8%	0.7840s	0s	nqueens

Слика 3: Извештај рада анализатора

Извештај садржи листу тока извршења програма са означеним петљама (слика 3). Функција *setQueen()* се рекурзивно позива, те се понавља више пута на овом списку. Прегледајући колону укупног извршења функције, увиђамо да ова функција заузима прилично много времена од укупног трајања извршења програма. Очигледно је да је управо она кандидат за паралелну обраду.

**НАПОМЕНА:** Информације које се прикупе овом анализом зависе од улазних параметара програма, то јест важе само за конкретну инстанцу извршавања. У случају овог програма, промена величине табле (што је улазни параметар) би само изменила укупна времена, али би релативно гледано закључак да функција *setQueen()* узима највише времена од укупног извршавања програма остао непромењен. То је зато што код овог програма путања извршавања остаје иста, само се мења број итерација и дубина рекурзије. Међутим, код неких других програма сасвим је могуће да се добију потпуно другачији резултати за различите улазне параметре.

### Обележавање критичних места у програмском коду (*Advisor annotations*)

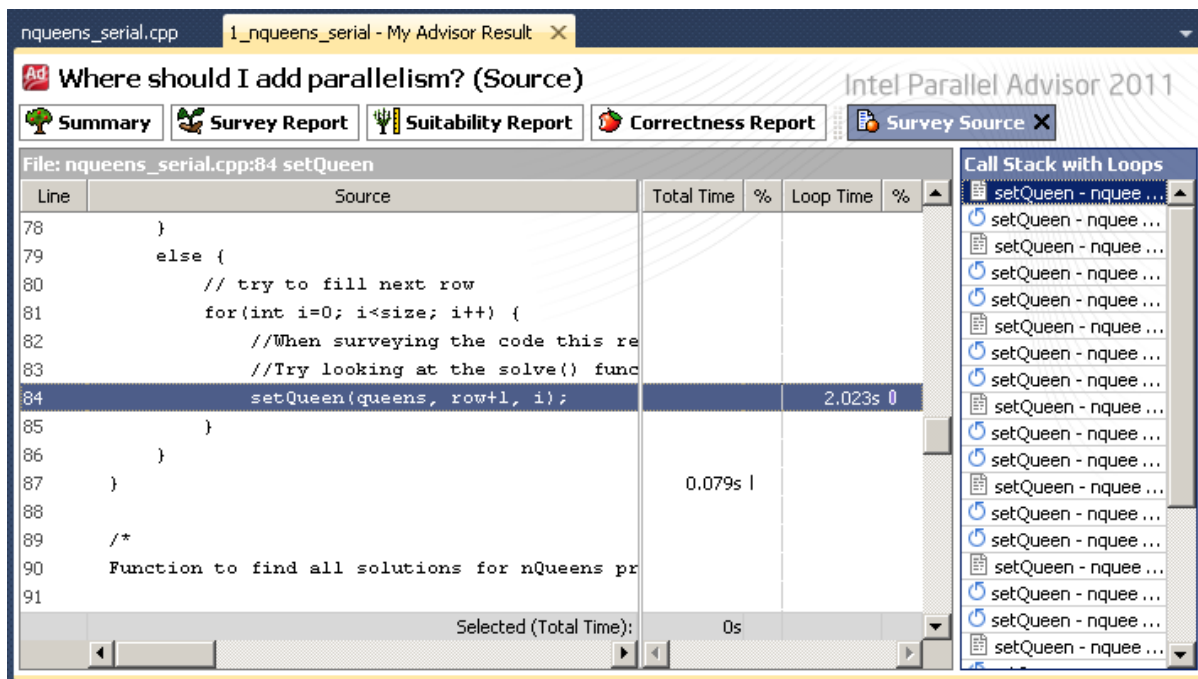
Користећи сакупљене податке из претходне анализе одабраћемо погодна места за паралелну обраду у програмском коду, тако што ћемо убацивати такозване знаке

(*annotations*), то јест обележићемо погодна места програмским исказима које разуме Адвајзор. Ова фаза се састоји из три корака:

1. Приказивање програмског кода у прозору *Survey Source*
2. Проналажење локација за паралелну обраду
3. Обележавање локација за паралелну обраду коришћењем назнаке места паралелизације и паралелног задатка (*parallel site and parallel task*).

### Приказивање програмског кода у прозору *Survey Source*

Двоструким притиском левог тастера миша на име функције *setQueen()*, или притиском десног тастера и одабиром опције *View Source* у прозору *Survey Report*, приказаће се прозор *Survey Source* (слика 4).



Слика 4: Прозор *Survey Source*

У том прозору са десне стране се налазе подаци о времену које је утрошено на извршавање кода. Још једном видимо да рекурзивна функција *setQueen()* користи готово све процесорско време приликом извршавања овог програма. Такође можемо видети време и за појединачне линије.

### Проналажење локација за паралелну обраду

Циљ нам је да доделимо обраду често извршаваних делова програмског кода различитим задацима који се могу извршавати истовремено. Осим проналажења дела који

троши највише времена, морамо анализирати и програмски ток до тог места, у нашем случају, функције *setQueen()*. Прегледом тела главне функције *main()* уочавамо да се пре позива функције *setQueen()* обрађују улазни параметари, иницијализују низови и позива функција *solve()*. Функција *solve()* позива функцију *setQueen()*, која се затим позива рекурзивно.

Двоструким притиском левог тастера на жељену линију отвориће се прозор са оригиналним кодом програма. Видимо да се он налази у функцији *solve*. На том месту треба коришћењем назнака да обележимо место паралелизације и паралелне задатке. Следеће поглавље објашњава назнаке за место паралелизација и назнаке за паралелни задатак.

### **Обележавање локација за паралелну обраду користећи назнаке места паралелизације и паралелног задатка**

Назнаке су у својој форми искази које саопштавају програмском алату Адвајзор програмерове намере у вези са паралелизацијом кода. Конкретно, имплементиране су као макрои који се свде на позив функција. Те функције не мењају кориснички код, већ само омогућавају програмском алату да прикупи одређене информације. Информације се прикупљају приликом извршавања програма у одговарајућем режиму. Важно је приметити да се те информације прикупљају приликом **секвенционалног** извршавања програма, јер никакав паралелизам још увек није уведен. Осим тога, и у овом случају важи напомена о зависности добијених информација (и на основу њих изведених података) од улазних параметара програма.

На основу прикупљених информација Адвајзор процењује ефекте које би паралелно извршавање у складу са датим знацима имало. Алат даје своју процену фактора убрзања, али и указује на места код којих би се могли јавити проблеми паралелизације (као што су потреба за синхронизацијом, регулисање искључивог приступа и слично). Овакав приступ је користан у фази избора најповољнијих делова кода који би се паралелизовали, као и начина паралелизације, зато што се знацима лако манипулише и њихово коришћење је далеко једноставније него право увођење паралелизма. Када се на основу процене одреде места и начини паралелизације, може се прећи на фазу увођења паралелизације.

Две врсте назнака су кључне за рад са Адвајзор алатом. То су:

- Назнаке места паралелизације, којима се обележава почетак и крај дела програма у који се жели увести паралелизација. Синтакса ових назнака је:  
ANNOTATE\_SITE\_BEGIN(jedinstveni\_naziv),  
ANNOTATE\_SITE\_END(jedinstveni\_naziv)

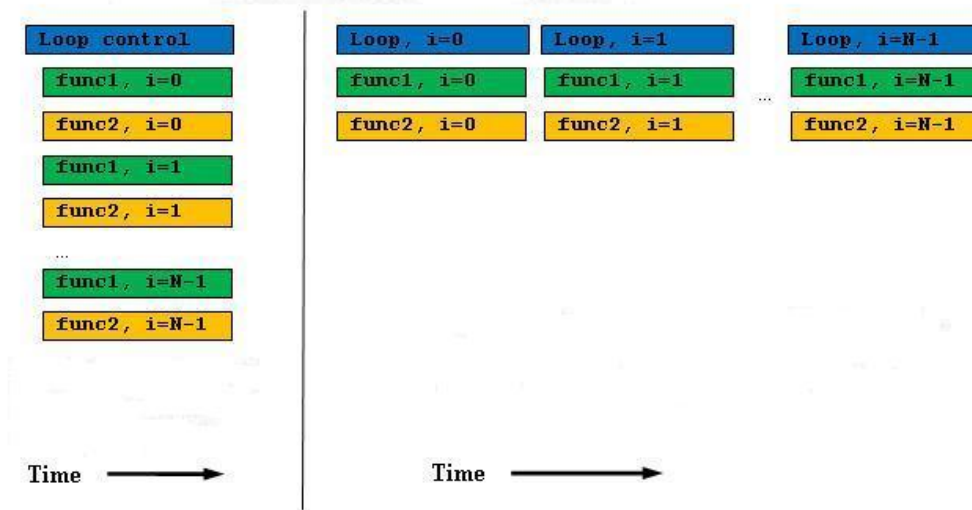
- Назнаке паралелног задатка, којима се обележава почетак и крај дела програма који представља један задатак. Синтакса ових назнака је:  
ANNOTATE\_TASK\_BEGIN(jedinstveni\_naziv),  
ANNOTATE\_TASK\_END(jedinstveni\_naziv)

**НАПОМЕНА:** Да би се код у који су додате назнаке уопште превео, потребно је укључити датотеку *advisor-annotate.h*.

Однос места паралелизације и паралелног задатка могао би се формулисати овако: Паралелни задатак представља део кода који се може извршавати у паралели са неким другим кодом, док место паралелизације дефинише који је то други код. Или: паралелни задатак се може извршавати у паралели само са задацима који припадају истом месту паралелизације. Следећи пример илуструје коришћење поменутих назнака:

<pre>... ANNOTATE_SITE_BEGIN(mesto); for (i=0; i&lt;N; i++) {     ANNOTATE_TASK_BEGIN(zadatak1);     func1(i);     ANNOTATE_TASK_END(zadatak1);     ANNOTATE_TASK_BEGIN(zadatak2);     func2(i);     ANNOTATE_TASK_END(zadatak2); } ANNOTATE_SITE_END(mesto); ...</pre>	<pre>... for (i=0; i&lt;N; i++) {     ANNOTATE_SITE_BEGIN(mesto);     ANNOTATE_TASK_BEGIN(zadatak1);     func1(i);     ANNOTATE_TASK_END(zadatak1);     ANNOTATE_TASK_BEGIN(zadatak2);     func2(i);     ANNOTATE_TASK_END(zadatak2);     ANNOTATE_SITE_END(mesto); } ...</pre>
---	---

Најпре уочимо да ће током извршавања датог парчета кода, функције 1 и 2 (које су назначене као задатак 1 и 2) бити позване N пута. У коду који је дат са леве стране назначена је програмова жеља да се свих N позива сваке од функција извршава у паралели. Са десне стране је код у којем је назначено да се у паралели извршавају само позиви функција у оквиру једне итерације петље, док сама петља итеририра секвенцијално. Такво понашање илустровано је на слици 5.



Слика 5: Илустрација разлике у извршавању претходно датих примера

**Питање:** Како треба назначити код у функцији *solve*, у нашем примеру? (За одговор погледајте пројекат **2\_nqueens\_annotated**.)

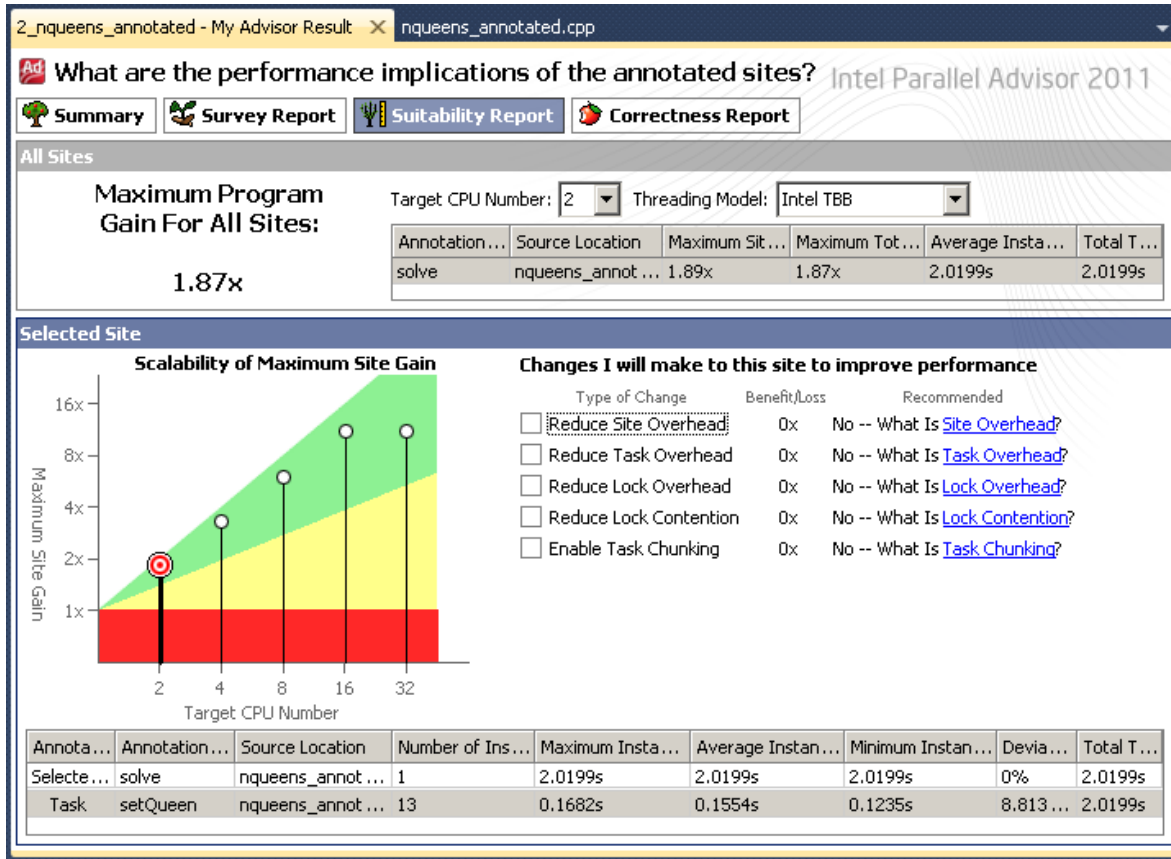
## Процена могућег убрзања паралелизацијом

У овој фази покренућемо алгоритам за процену ефеката паралелизације са становишта убрзања и анализираћемо добијене резултате.

Најпре треба поставити пројекат **2\_nqueens\_annotated**, као стартни пројекат. Тај пројекат садржи већ назначени програмски код из првој пројекта.

### Покретање анализе за процену убрзања

Анализу покрећемо притиском на дугме **Start** у оквиру корака три (Check Suitability) у прозору **Advisor Workflow**. И овога пута треба да покренемо **Release** конфигурацију. Након што се програм изврши у режиму за прикупљање података релевантних за ову анализу, подаци ће бити приказани у оквиру извештаја рада (слика 6).



Слика 6: Извештај рада анализатора ефеката паралелизма

Извештај даје процену убрзања изражену релативно у односу на време секвенциалног извршавања. Потенцијално убрзање зависи и од неколку параметара које је могуће у извештају мењати. Један од параметара је и број процесорских језгара. На графику који приказује потенцијално убрзање у зависности од броја језгара, види се да фактор убрзања расте скоро линеарно све до 16 језгара, након чега стагнира.

**ПИТАЊЕ:** Зашто је то тако? (Помоћно питање 1: Како би изгледао график да је резолуција на икс оси већа, то јест да је дата вредност за све могуће количине процесорских језгара? Помоћно питање 2: Које се напомене дате у овој вежби односе на ову анализу?)

Посебну пажњу треба обратити на табелу у дну извештаја. Врло корисна информација у тој табели је просечно време извршавања задатка, које не би требало да буде блиско времену потребном за стварање и уништавање програмске нити (што је реда величине  $10^{-5}$  секунди). У нашем случају просечно време извршавања нити је 0,1714 секунде, што је довољно много.

Када смо задовољни процењеним убрзањем прелазимо на фазу провере ваљаности предложеног концепта паралелизације.



## Провера ваљаности паралелизације

У претходној фази дали смо предлог паралелизације и проценили њену ефикасност. Међутим, урадили смо то не узимајући у обзир проблеме који се јављају код кода који се паралелно извршава, а не постоје код секвенцијалног кода. У овој фази покренућемо алгоритам који покушава да укаже управо на такве проблеме.

### Покретање анализе ваљаности

Као и претходне анализе, и ова анализа подразумева извршавање програма у одговарајућем режиму. Но, у овом случају анализа успорава извршавање од 50 до неколико стотина пута. Због тога је потребно посебно повести рачуна о улазним параметрима програма, не би ли се време извршавања свело на разумну меру. У нашем контретном случају једини улазни параметар је величина табле, и он директно утиче на време извршавања. Подразумевана величина табле (13) је коришћена у претходним анализама, али сада би ваљало да је смањимо на, рецимо, 8. Параметри командне линије се задају на следећем месту *Configuration Properties > Debugging > Command Arguments*.

**НАПОМЕНА:** За разлику од претходне две анализе, ову анализу је потребно покренути за **Debug** конфигурацију програма. Сама чињеница да се програм покреће у Debug конфигурацији је делимичан извор успорења које се при овој анализи јавља.

Покретање анализе је исто као и у претходним корацима: притиском на дугме **Start** у оквиру корака четири (Check Correctness) у прозору **Advisor Workflow**. Након што се програм изврши у режиму за прикупљање података релевантних за ову анализу, биће отворен извештај рада (слика 7).

2\_nqueens\_annotated - My Advisor Result x nqueens\_annotated.cpp

Did the annotated tasks expose data sharing problems? Intel Parallel Advisor 2011

Summary Survey Report Suitability Report Correctness Report

ID	Problem	Sources	Modules	State
P1	Memory reuse	newaop.cpp; nqueens_annotated.cpp	2_nqueens_annotated.exe	Not fixed
P2	Data communication	nqueens_annotated.cpp	2_nqueens_annotated.exe	Not fixed
P3	Parallel site information	nqueens_annotated.cpp	2_nqueens_annotated.exe	Information

Memory reuse: Observations

ID	Description	Source	Function	Module	State
X3	Allocation site	newaop.cpp:6	new[]	2_nqueens_annotated.exe	Information
<pre>4 void * __CRTDECL operator new[](size_t count) _THROW1(std::bad_alloc) 5 { // try to allocate count bytes for an array 6   return (operator new(count)); 7 } 8</pre>					
X4	Parallel site	nqueens_annotated.cp... solve		2_nqueens_annotated.exe	Information
<pre>98 int * queens = new int[size]; //array representing queens placed on a che 99 100 ANNOTATE_SITE_BEGIN(solve); 101 for(int i=0; i&lt;size; i++) { 102   // try all positions in first row </pre>					
X5	Write	nqueens_annotated.cp... setQueen		2_nqueens_annotated.exe	Not fixed
<pre>75 } 76 // column is ok, set the queen</pre>					

Filter

Severity

Error	2 items
Remark	1 item

Problem

Data communication	1 item
Memory reuse	1 item
Parallel site information	1 item

Source

newaop.cpp	1 item
nqueens_annotated.cpp	3 items

Module

2_nqueens_annotated.exe	3 items
-------------------------	---------

State

Information	1 item
Not fixed	2 items

Sort By Item Name

Слика 7: Извештај рада анализе ваљаности

Анализа у суштини проналази критичне секције које би постојале у паралелном програму формираном у складу са назнакама. Уз то, анализа помаже код одређивања врсте проблема који се могу јавити у вези са одређеном критичном секцијом.

У извештају се види да је анализа пронашла два проблема, то јест грешке. Један је проблем конкурентног приступа подацима (енг. data sharing problem), а други је проблем коришћења исте меморије (енг. memory reuse problem). Списак грешака и информације о њима је сличак списку грешака који се јавља приликом превођења програма. Разлика је ипак у томе што код грешака приликом превођења једна грешка је везана за једно место у коду, док се грешке које се јављају код паралелног програмирања односе на више локација у коду. Тако када изаберете грешку у вези са приступом подацима (data sharing) у доњем делу прозора са извештајем видећете три програмске локације (слика 8).

Data communication: Observations					
ID	Description	Source	Function	Module	State
X8	Parallel site	nqueens_annotated.cpp:100	solve	2_nqueens_annotated.exe	Information
<pre>98     int * queens = new int[size]; //array representing queens placed on a chess board.  Index is row position, va 99 100     ANNOTATE_SITE_BEGIN(solve); 101     for(int i=0; i&lt;size; i++) { 102         // try all positions in first row</pre>					
X9	Write	nqueens_annotated.cpp:80	setQueen	2_nqueens_annotated.exe	Not fixed
<pre>78 79     if(row==size-1) { 80         nrOfSolutions++; //Placed final queen, found a solution! 81     } 82     else {</pre>					
X10	Read	nqueens_annotated.cpp:80	setQueen	2_nqueens_annotated.exe	Not fixed
<pre>78 79     if(row==size-1) { 80         nrOfSolutions++; //Placed final queen, found a solution! 81     } 82     else {</pre>					

Слика 8: Информације о проблему конкурентног приступа подацима

Прво парче кода које је приказано је почетак места паралелизације на које се ова грешка односи. Наредно парче кода приказује место где се одређена променљива уписује, док последње парче кода приказује место где се та променљива чита. У овом конкретном случају те две локације су исте, међутим оне су део паралелног задатка, дакле извршавају се у различитим програмским нитима и зато представљају критичну секцију. Чињеница да у исту променљиву једна нит уписује вредност, док је друга нит чита, сигнал је да је потребно регулисати приступ тој променљивој, било путем увођења синхронизације, било обезбеђивањем искључивог приступа. Сада када смо свесни тога, потребно је да то саопшtimo и алату, како би алгоритам анализе знао да намеравамо регулисати приступ тој променљивој. У ту сврху користимо следећи пар назнака браве за регулисање приступа: `ANNOTATE_LOCK_ACQUIRE(adresa)` и `ANNOTATE_LOCK_RELEASE(adresa)`, где је аргумент макроа адреса променљиве која се штити или адреса неке променљиве која представља браву.

Memory reuse: Observations					
ID	Description	Source	Function	Module	State
X3	Allocation site	nqueens.cpp:6	new[]	2_nqueens_annotated.exe	Information
<pre>4 void *__CRTDECL operator new[](size_t count) _THROW1(std::bad_alloc) 5 { // try to allocate count bytes for an array 6   return (operator new(count)); 7 } 8</pre>					
X4	Parallel site	nqueens_annotated.cpp:100 solve		2_nqueens_annotated.exe	Information
<pre>98 int * queens = new int[size]; //array representing queens placed on a chess board. Index is row position, va 99 100 ANNOTATE_SITE BEGIN(solve); 101 for(int i=0; i&lt;size; i++) { 102   // try all positions in first row</pre>					
X5	Write	nqueens_annotated.cpp:77 setQueen		2_nqueens_annotated.exe	Not fixed
<pre>75 } 76 // column is ok, set the queen 77 queens[row]=col; 78 79 if(row==size-1) {</pre>					
X6	Write	nqueens_annotated.cpp:77 setQueen		2_nqueens_annotated.exe	Not fixed
<pre>75 } 76 // column is ok, set the queen 77 queens[row]=col; 78 79 if(row==size-1) {</pre>					
X7	Read	nqueens_annotated.cpp:68 setQueen		2_nqueens_annotated.exe	Not fixed
<pre>66 for(int i=0; i&lt;row; i++) { 67   // vertical attacks 68   if (queens[i]==col) { 69     return; 70 }</pre>					

Слика 9: Информације о проблему коришћења исте меморије

Извештај у вези са проблемом коришћења исте меморије је нешто сложенији. Обухвата 5 програмских локација (слика 9). Кренимо од друге локације (обележене са X4). Као и код претходног проблема, то парче кода представља почетак места паралелизације на које се овај проблем односи. Наредна два парчета представљају места где се уписује у меморију, док последње парче приказује читање из меморије. Вратимо се напоскон на прву локацију. Она приказује место где се заузима меморија која се користи од стране различитих нити (*Allocation site*). Међутим, приказан је код у системској функцији за заузимање меморије, те се не види јасно о којој се меморији ради. Дакле, потребно је отићи један корак назад у програмском стеку. Да бисмо то постигли потребно је да дуплим кликом на прву локацију отворимо детаљнији приказ изворног кода који се тиче овог проблема (слика 10). У горњем делу детаљног приказа, десно од програмског кода видимо програмски стек по којем се можемо кретати. На првом следећем нивоу стека видимо да је меморија о којој се ради заправо низ целобројних вредности који се у линији 98 додељује показивачу **queens** (додуше, то се могло сазнати и гледањем програмских локација уписивања и читања меморије, међутим у неким сложенијим примерима то не мора бити тако очигледно).

Did the annotated tasks expose data sharing problems? (Source)

Intel Parallel Advisor 2011

Summary Survey Report Suitability Report Correctness Report Correctness Source

Focus Observation: newaop.cpp:6 - Allocation site

```
1 // newaop -- operator new[](size_t) REPLACEABLE
2 #include <new>
3
4 void *__CRTDECL operator new[](size_t count) _THROW1(std::bad_alloc)
5 { // try to allocate count bytes for an array
6   return (operator new(count));
7 }
8
9 /*
10  * Copyright (c) 1992-2007 by P.J. Plauger. ALL RIGHTS RESERVED.
11  * Consult your license regarding permissions and restrictions.
12  * VS.03:0009 */
```

Call Stack

- new[] - newaop.cpp:6
- solve - nqueens\_annotated.cpp:98
- main - nqueens\_annotated.cpp:142
- \_mainCRTStartup - crtexe.c:555
- mainCRTStartup - crtexe.c:370

Related Observation: nqueens\_annotated.cpp:77 - Write

```
72 if (abs(queens[i]-col) == (row-i) ) {
73   return;
74 }
75
76 // column is ok, set the queen
77 queens[row]=col;
78
79 if(row==size-1) {
80   nrOfSolutions++; //Placed final queen, found a solution!
81 }
82 else {
83   // try to fill next row
```

Call Stack

- setQueen - nqueens\_annotated.cpp:77
- solve - nqueens\_annotated.cpp:105
- main - nqueens\_annotated.cpp:142
- \_mainCRTStartup - crtexe.c:555
- mainCRTStartup - crtexe.c:370

Memory reuse: Observations

ID	Description	Source	Function	Module	State
X3	Allocation site	newaop.cpp:6	new[]	2_nqueens_annotated.exe	Information
X4	Parallel site	nqueens_annotated.cp...	solve	2_nqueens_annotated.exe	Information
X5	Write	nqueens_annotated.cp...	setQueen	2_nqueens_annotated.exe	Not fixed
X6	Write	nqueens_annotated.cp...	setQueen	2_nqueens_annotated.exe	Not fixed
X7	Read	nqueens_annotated.cp...	setQueen	2_nqueens_annotated.exe	Not fixed

Relationship Diagram

```
graph LR
    A[Allocation site  
newaop.cpp:6] --> B[Write  
nqueens_anno]
    B --> C[Write  
nqueens_annot]
```

Слика 10: Детаљнији приказ програмског кода који се тиче проблема пронађених при анализи ваљаности паралелизације

Меморија на коју показивач **queens** показује користи се као радна меморија у коју се уписују међурезултати са раличитих нивоа рекурзивног позивања функције **setQueen**. Док се петља у функцији **solve** извршавала секвенцијално та иста меморија се могла поново искористити за све итерације петље. Међутим, када се итерације петље извршавају конкурентно, свака нит, то јест сваки задатак, мора имати своју меморију. Због тога је решење овог проблема пребацивање места алокације меморије у део паралелног задатка.

Након што смо исправили оба проблема добро би било да анализу ваљаности паралелизације поново покренемо. Овога пута не би требало да буде никаквих проблема.

**НАПОМЕНА:** Анализа ваљаности ради над секвенцијалним програмом прикупљајући потребне податке на местима где су назнаке. Због тога што се програм не извршава заиста паралелно оваква анализа не гарантује да ће открити све проблеме. Основни случај када ова анализа не пријављује проблем је када се конкурентан притуп променљивој унутар паралелног задатка дешава у истој функцији у којој је назначено место паралелизације. Ако је дотична променљива локална за ту функцију анализа не

може разлучити да ли је досег променљиве шири или ужи од паралелног задатка, јер је досег променљиве цела функција, ма где променљива била дефинисана. Ако је досег променљиве ужи (променљива се дефинише унутар паралелног задатка) онда нема проблема са конкурентним приступом подацима, док ако је досег шири (променљива се дефинише пре паралелног задатка) онда има проблема. Сматрајући да је боље не пријавити грешку у неком случају него пријавити лажну грешку, аутори ове анализе су одлучили да она у описаном случају не пријављује ништа.

## Увођење паралелизма

Последњи корак у процесу паралелизације полазног програма је коначно увођење паралелизма у код коришћењем неког од програмских оквира за паралелизацију. У овој вежби обрадићемо два најчешће коришћена оквира иза којих стоји Интел.

### Intel TBB – „Градивни елементи конкурентних програма“

Први програмски оквир је библиотека Градивни елементи конкурентних програма, или скраћено TBB (енг. *Threading Building Blocks*). Та библиотека садржи C++ шаблон класе у чијим изведбама је садржано руковање нитима, и самим тим паралелиним задацима. На тај начин детаљи у вези са руковањем нитима су скривени од корисника, који због тога може радити на вишем нивоу апстракције. У самом називу библиотеке, „градивни елементи“, наговештен је очекивани начин на који би се библиотека требало користити. Библиотека нуди елементе који осликавају неколико типичних случајева паралелизације, а корисник, након што препозна на који типичан случај се своди његов конкретан случај, коришћењем одговарајућих елемената пише паралелни код.

Први задатак је паралелизовати место паралелизације назначено у функцији **solve**. У тој функцији назначено је да свака итерација петље треба да се извршава као један паралелни задатак, тј. као посебна нит. Један од елемената библиотеке који одговара оваквом случају паралелизације је *parallel\_for* елемент. У изворној датотеци *nqueens\_tbb.cpp* у оквиру пројекта *3\_nqueens\_tbb* можемо видети како изгледа једна од могућих употреба тог елемента. Елементу *parallel\_for* потребно је проследити две ствари: опсег вредности кроз које треба итерирати и функцију која представља тело петље, то јест тело нити.

**ПИТАЊЕ:** Зашто је битно да је у питању **for** петља? По чему се **for** петља разликује од осталих врста петљи?

Опсег вредности се задаје објектом *blocked\_range* шаблон класе, која је исто елемент TBB библиотеке. Шаблон класа се конструише над типом итератора; у нашем случају то је **size\_t** (може бити и обичан **int** тип, али је тип **size\_t** уобичајен за целобројне итераторе зато што је неозначен и његова величина одговара простору итерације који је

могућ на датој платформи). Као аргументи конструктора се прослеђују почетна вредност, крајња вредност и корак.

Очекује се да је функција за тело петље дата у следећој форми:

```
class naziv
{
public:
    void operator() (const blocked_range<size_t>& r) const
    {
        for (size_t i = r.begin(); i != r.end(); ++i)
        {
            //тело петље
        }
    }
};
```

Може се приметити да је аргумент ове функције управо објекат **blocked\_range** шаблон класе. У самом телу функције обавезно је присутна петља која итерира кроз опсег дефинисан аргументом. Тек тело те петље представља заправо тело спољне петље. Дакле, ефективно имамо две угњежене **for** петље.

**ПИТАЊЕ:** Зашто имамо две угњежене петље? (Помоћно питање 1: Вратити се на слику 7, шта све утиче на брзину извршавања? Помоћно питање 2: У каквом су односу број процесорских језгара и број итерација петље?)

Други задатак је регулисање приступа променљивој **nrOfSolutions**. За потребе регулисања приступа ТВВ библиотека обезбеђује класичан механизам заштите у виду објеката искључивог приступа, илити мутекса (енг. mutex – **mutually exclusive**). Библиотека нуди неколико различитих мутекса, са различитим карактеристикама. У изворној датотеци **nqueens\_tbb.cpp** употребљен је **spin\_mutex** који је погодан за кратке критичне секције. Заузимање и ослобађање објекта искључивог приступа се остварује кроз конструктор и деструктор објекта **scoped\_lock**, којем је као аргумент прослеђен дотични објекат искључивог приступа.

### Intel Cilk Plus

Други програмски оквир за паралелизацију је програмски језик Cilk. Синтакса програмског језика Cilk се само у неколико детаља разликује од синтаксе језика C++ тако да се Cilk може сматрати и проширењем C++-а. За превођење програма написаних у Cilk-у потребан је компајлер који уме да преведе Cilk синтаксу. То је битна разлика у односу на ТВВ оквир, јер паралелне програме написане коришћењем ТВВ библиотеке може превести било који C++ компајлер; довољно је имати библиотеку. Интелов компајлер који долази уз Parallel Studio преводи Cilk код. Када погледате сва четири пројекта у овој вежби, видећете да је подешено да се само пројекат 3\_nqueens\_cilk преводи Интеловим компајлером, док се остали пројекти преводи Мајкрософтовим C++ компајлером.

Предност Cilk-а уочава се код првог задатка при паралелизацији нашег програма. Све што је потребно урадити да би се петља паралелизовала је ставити резервисану реч Cilk језика **cilk\_for** уместо C++-овског **for**. Компајлер ће ту петљу претворити у код који се извршава паралелно.

За регулисање приступа дељеној променљивој језик Cilk сам по себи не нуди објекте искључивог приступа. Могу се користити било који објекти искључивог приступа, укључујући и објекте искључивог приступа из TBV библиотеке. Међутим, при програмирању у Cilk-у намеће се коришћење такозваних редуктора. Редуктори су посебне променљиве над којима су омогућене само асоцијативне операције. Пошто су операције асоцијативне, редослед извршавања је небитан и крајњи резултат је ваљан докле год се све операције изврше. У том смислу редуктори су имплементирани тако да свака нит приступа свом „примерку“ редукторске променљиве, па због тога нема потребе за искључивим приступом. На крају се само резултат свих операција срачуна, то јест „редукује“ на крајњу вредност.

У изворној датотеци **nqueend\_cilk.cpp** за променљиву **nrOfSolutions** корићен је сабирајући редуктор **reducer\_opadd** јер је над њиме потребно вршити операцију сабирања (тачније, увећања за један). Над сабирајућим редуктором су дозвољене операције самоувећања (**+=**), самоумањења (**-=**), инкрементовања (**++**) и декрементовања (**--**). Вредност редуктора се добија позивањем методе **get\_value** која срачунава вредност на основу обављених операција, ако вредност већ није срачуната.