Thomas Fahle

HowTo

Parallel::Iterator - Mehrere Tasks parallel ausführen

`Parallel::Iterator` von Andy Armstrong erlaubt die gleichzeitige (parallele) Ausführung mehrerer Tasks auf einem Rechner mit (vorzugsweise) mehreren CPUs.

Dabei kümmert sich Parallel::Iterator um korrekte Interprozeß-Kommunikation, Forking und Verteilung auf mehrere CPUs, so daß man sich voll und ganz auf die eigentliche Aufgabe konzentrieren kann.

Anwendungsgebiete

Da der Forking-Prozess einen nicht zu unterschätzenden Overhead mit sich zieht, bietet sich die Verwendung von `Parallel::Iterator` vor allem bei folgenden Aufgaben an:

- Anwendungen, die oft oder lange auf IO/Netzwerk warten
- CPU-intensive Kalkulationen, die sich (einfach) auf mehrere CPUs verteilen lassen

Iteratoren, Tasks und Worker

`Parallel::Iterator` verwendet Iteratoren, Worker und Tasks. Ein Worker ist eine Subroutine, welche die eigentliche Arbeit ausführt - Tasks sind Listen der Parameter, die an den Worker übergeben werden - Iteratoren gehen durch die Taskliste und parallelisieren den Worker.

Neben der Möglichkeit eigene Iteratoren zu definieren, bietet `Parallel::Iterator` bereits zwei fertige Iteratoren - `iterate_as_array` und `iterate_as_hash` - an, die eine Referenz auf ein Unterprogramm (Worker) und eine Referenz auf die Liste der Tasks entgegen nehmen.

In dem folgenden einfachen und nicht wirlich praxisrelevantem Beispiel wird eine Liste von Quadratwurzeln aus einer Liste von Zahlen parallel ermittelt.

iterate_as_array

Der Worker `square_root` erhält als ersten Parameter den Index aus `@numbers` (Tasks) und als weiteren Parameter die zu bearbeitende Zahl.

Um die Ergebnisse im `@output` korrekt anzuordnen, müssen Index **und** Ergebnis aus `square_root` zurückgegeben werden.

Falls die Reihenfolge der Ergebnisse keine Rolle spielt, kann der Index in der Rückgabeliste entfallen.

```
#!/usr/bin/perl
use warnings;
use strict;

use Parallel::Iterator qw(iterate_as_array);

# Tasks
       # Indizes: 0  1  2  3  4  5  6
my @numbers = qw/ 1  4  9  16 25 36 49 /;

my @output = iterate_as_array(
    \&square_root,
    \@numbers,
);

print join("\t", @output), "\n";

# Worker
sub square_root {
   my ($index, $number) =  @_;
   my $sr = sqrt($number);

   # return index and value
   return ($index,$sr);
}
```

iteratate_as_hash

Das Unterprogramm `square_root` (worker) erhält als ersten Parameter den Key aus `%numbers` (Tasks) und als weiteren Parameter die zu bearbeitende Zahl.

Um die Ergebnisse im `%output` korrekt zuzuordnen, müssen Schlüssel **und** Ergebnis aus `square_root` zurückgegeben werden.

```
#!/usr/bin/perl
use warnings;
use strict;

use Parallel::Iterator qw(iterate_as_hash);

# Tasks
my %numbers = (
   1  => 1,
   4  => 4,
   9  => 9,
   16 => 16,
   25 => 25,
   36 => 36,
   49 => 49,
);

my %output = iterate_as_hash(
    \&square_root,
    \%numbers,
);

my @numbers = sort { $a <=> $b }keys %output

foreach my $number ( @numbers ) {
    print "The square root of $number is "
          . "$output{$number}\n";
}

# Worker
sub square_root {
   my ( $key, $number ) = @_;
   my $sr = sqrt($number);

   # return key and value
   return ( $key, $sr );
}
```

Optionen und Optimierungen

Die beiden dargestellten Iteratoren werden mit sinvollen Performance-Vorgabewerten ausgeliefert. Individuelles Tuning der Iteratoren ist durch Optionen, deren Erläuterung den Rahmen dieser Einführung deutlich sprengen würde, möglich.

Beispiel Link-Checker

Eine typische Anwendung, die kaum Prozessorzeit erfordert, aber oft lange auf IO wartet, ist das Holen von Webseiten.

In diesem einfachen Beispiel wird der HTTP-Statuscode verschiedener Websites ermittelt.

Über die Option `workers` wird die Anzahl der parallelen Tasks begrenzt.

```
#!/usr/bin/perl
use strict;
use warnings;

use LWP::UserAgent;
use Parallel::Iterator qw/iterate_as_array/;

#  a list of pages to fetch
my @urls = qw(
  http://www.perl-howto.de
  http://www.perl.org
  http://www.yahoo.de
  http://www.google.de
  http://www.tagesschau.de
  http://www.zdf.de
);

my $ua = LWP::UserAgent->new();

# this worker fetches a page and returns
# the HTTP status code
my $worker = sub {
    my $index    = shift;
    my $url      = shift;
    my $response = $ua->get($url);
    return ( $index, $response->code() );
};

# Number of parallel tasks
my %options = ();
$options{workers} = 2;

# Fetch pages in parallel
my @status_codes = iterate_as_array(
\%options, $worker, \@urls );

# Display results
my %codes = ();

# Hash slice
@codes{@urls} = @status_codes;

# output results
my $format = "%-40s %s\n";
printf( "$format", 'URL', 'Status' );
foreach my $url ( sort keys %codes ) {
    printf( "$format", $url, $codes{$url} );
}
__END__
```