nking Parallel

cks was designed to make expressing parallelism much easier
tails and providing strong support for the best ways to pro-
Here is a quick overview of how Threading Building Blocks
will define and review in this chapter:

pose your problem into concurrent tasks (tasks that can run

em so that there are enough concurrent tasks to keep all the
sy while minimizing the overhead of managing the parallel

nology underlying the concurrency in programs—and how
by Threading Building Blocks so that you can just focus on

ynchronization inherent in Threading Building Blocks helps
locks. If you still must use locks, there are special features for
ead Checker to find *deadlocks* and *race conditions*, which
volving locks.

utilize algorithms, from Chapters 3 and 4.

in improving performance. The Threading Build Blocks task
tuned for caches.

f tasks that can run at the same time (concurrent tasks), data
imize conflicts among tasks, and recursion.

nd ourselves thinking about parallelism. Here are a few

vait in a long line, you have undoubtedly wished there were
ter) lines, or multiple people at the front of the line helping
re quickly. Grocery store checkout lines, lines to get train
coffee, and lines to buy books in a bookstore are examples.

Lots of repetitive work

When you have a big task to do, which many people could help with at the same time, you have undoubtedly wished for more people to help you. Moving all your possessions from an old dwelling to a new one, stuffing letters in envelopes for a mass mailing, and installing the same software on each new computer in your lab are examples.

The point here is simple: parallelism is not unknown to us. In fact, it is quite natural to think about opportunities to divide work and do it in parallel. It just might seem unusual for the moment to program that way. Once you dig in and start using parallelism, you will Think Parallel. You will think first about the parallelism in your project, and only then think about coding it.

Decomposition

When you think about your project, how do you find the parallelism?

At the highest level, parallelism exists either in the form of data on which to operate in parallel, or in the form of tasks to execute concurrently. And these forms are *not* mutually exclusive.

Data Parallelism

Data parallelism (Figure 2-1) is easy to picture. Take lots of data and apply the same transformation to each piece of the data. In Figure 2-1, each letter in the data set is capitalized and becomes the corresponding uppercase letter. This simple example shows that given a data set and an operation that can be applied element by element, we can apply the same task concurrently to each element. Programmers writing code for supercomputers love this sort of problem and consider it so easy to do in parallel that it has been called *embarrassingly parallel*. A word of advice: if you have lots of data parallelism, do not be embarrassed—take advantage of it and be very happy. Consider it *happy parallelism*.

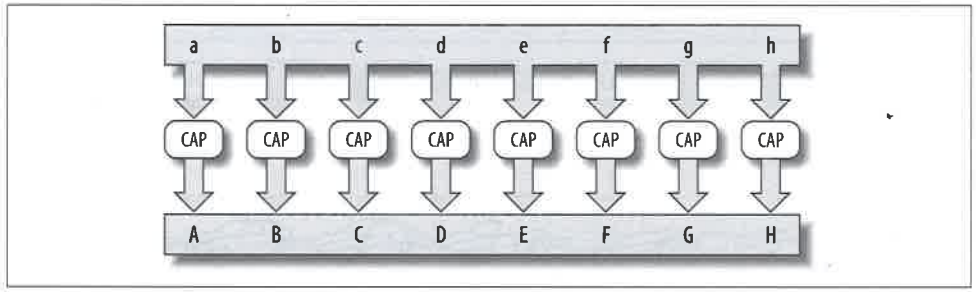


Figure 2-1. Data parallelism

Elements of Thinking Parallel

Threading Building Blocks was designed to make expressing parallelism much easier by abstracting away details and providing strong support for the best ways to program for parallelism. Here is a quick overview of how Threading Building Blocks addresses the topics we will define and review in this chapter:

Decomposition
: Learning to decompose your problem into concurrent tasks (tasks that can run at the same time).

Scaling
: Expressing a problem so that there are enough concurrent tasks to keep all the processor cores busy while minimizing the overhead of managing the parallel program.

Threads
: A guide to the technology underlying the concurrency in programs—and how they are abstracted by Threading Building Blocks so that you can just focus on your tasks.

Correctness
: How the implicit synchronization inherent in Threading Building Blocks helps minimize the use of *locks*. If you still must use locks, there are special features for using the Intel Thread Checker to find *deadlocks* and *race conditions*, which result from errors involving locks.

Abstraction and patterns
: How to choose and utilize algorithms, from Chapters 3 and 4.

Caches
: A key consideration in improving performance. The Threading Build Blocks task scheduler is already tuned for caches.

Intuition
: Thinking in terms of tasks that can run at the same time (concurrent tasks), data decomposed to minimize conflicts among tasks, and recursion.

In everyday life, we find ourselves thinking about parallelism. Here are a few examples:

Long lines
: When you have to wait in a long line, you have undoubtedly wished there were multiple shorter (faster) lines, or multiple people at the front of the line helping serve customers more quickly. Grocery store checkout lines, lines to get train tickets, lines to buy coffee, and lines to buy books in a bookstore are examples.

Lots of repetitive work
: When you have a big task to do, which many people co
time, you have undoubtedly wished for more people
your possessions from an old dwelling to a new one, st
for a mass mailing, and installing the same software o
your lab are examples.

The point here is simple: parallelism is not unknown to us.
to think about opportunities to divide work and do it in pa
unusual for the moment to program that way. Once you dig
lelism, you will Think Parallel. You will think first about
project, and only then think about coding it.

Decomposition

When you think about your project, how do you find the pa

At the highest level, parallelism exists either in the form of
in parallel, or in the form of tasks to execute concurrently.
mutually exclusive.

Data Parallelism

Data parallelism (Figure 2-1) is easy to picture. Take lots of
transformation to each piece of the data. In Figure 2-1, eac
capitalized and becomes the corresponding uppercase lett
shows that given a data set and an operation that can be app
we can apply the same task concurrently to each element. P
for supercomputers love this sort of problem and consider i
that it has been called *embarrassingly parallel*. A word of a
data parallelism, do not be embarrassed—take advantage
Consider it *happy parallelism*.

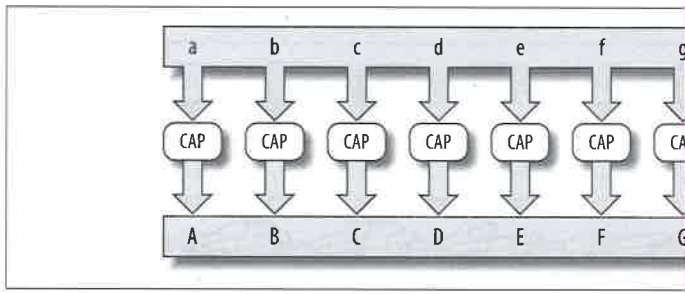


Figure 2-1. Data parallelism

Task Parallelism

Data parallelism is eventually limited by the amount of data you want to process, and your thoughts will then turn to task parallelism (Figure 2-2). Task parallelism means lots of different, independent tasks that are linked by sharing the data they consume. This, too, can be *embarrassingly parallel*. Figure 2-2 uses as examples some mathematical operations that can each be applied to the same data set to compute values that are independent. In this case, the average value, the minimum value, the binary OR function, and the geometric mean of the data set are computed.

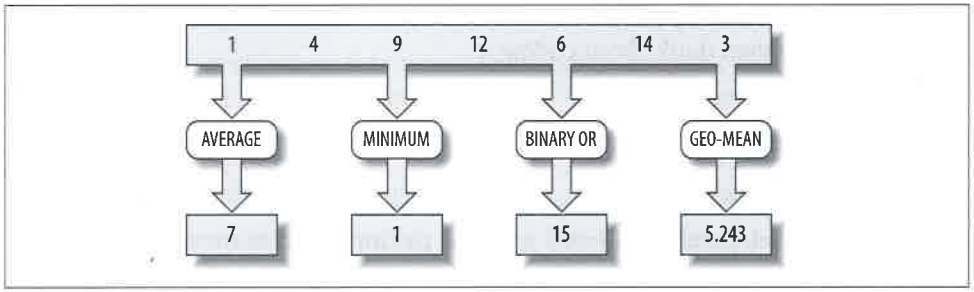


Figure 2-2. Task parallelism

Pipelining (Task and Data Parallelism Together)

Pure task parallelism is harder to find than pure data parallelism. Often, when you find task parallelism, it's a special kind referred to as *pipelining*. In this kind of algorithm, many independent tasks need to be applied to a stream of data. Each item is processed by stages as they pass through, as shown by the letter A in Figure 2-3. A stream of data can be processed more quickly if you use a pipeline because different items can pass through different stages at the same time, as shown in Figure 2-4. A pipeline can also be more sophisticated than other processes: it can reroute data or skip steps for chosen items. Automobile assembly lines are good examples of pipelines; materials flow through a pipeline and get a little work done at each step (Figure 2-4).

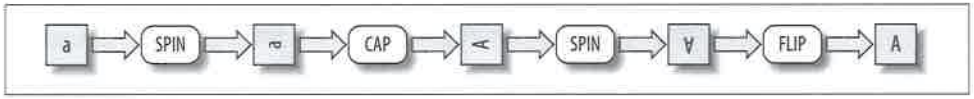


Figure 2-3. Pipeline

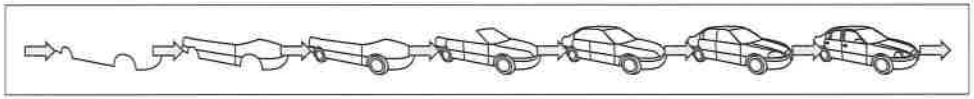


Figure 2-4. A pipeline in action with data flowing through it

Mixed Solutions

Consider the task of folding, stuffing, sealing, addressing, s
ters. If you assemble a group of six people for the task of
you can arrange each person to specialize in and perform o
ion (Figure 2-5). This contrasts with data parallelism, whe
and give a batch of everything to each person (Figure 2-6).
the steps on his collection of materials as his task.

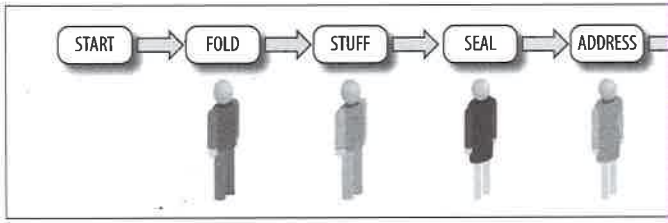


Figure 2-5. Pipelining—each person has a different job

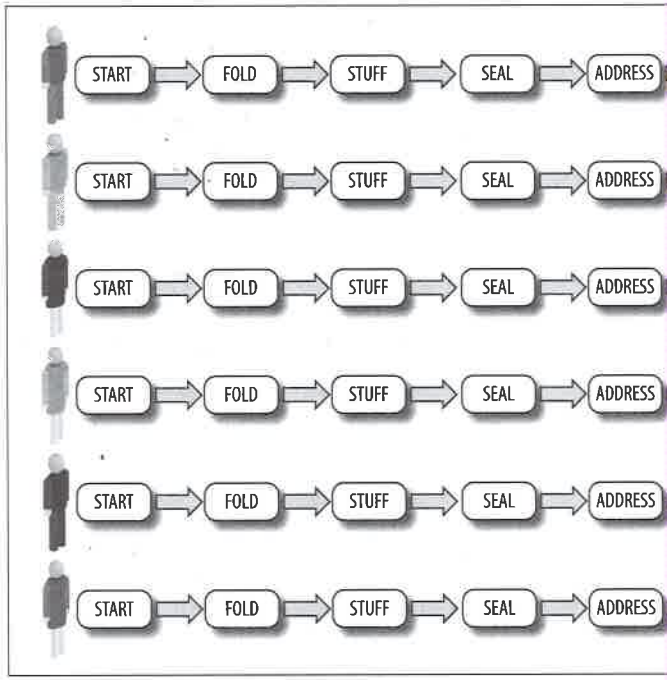
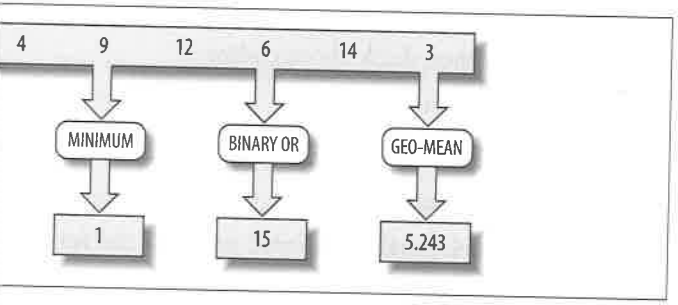


Figure 2-6. Data parallelism—each person has the same job

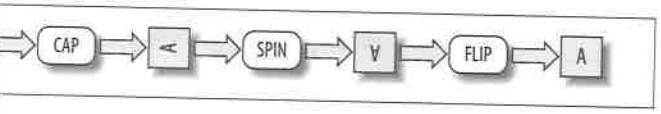
Figure 2-6 is clearly the right choice if every person has t
tion far from each other. That is called *coarse-grained* par
actions between the tasks are infrequent (they only
envelopes, then leave and do their task, including mailing

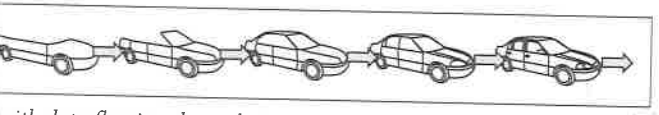
tually limited by the amount of data you want to process, then turn to task parallelism (Figure 2-2). Task parallelism independent tasks that are linked by sharing the data they be *embarrassingly parallel*. Figure 2-2 uses as examples some that can each be applied to the same data set to compute ent. In this case, the average value, the minimum value, the the geometric mean of the data set are computed.



Data Parallelism Together)

arder to find than pure data parallelism. Often, when you a special kind referred to as *pipelining*. In this kind of algo- tasks need to be applied to a stream of data. Each item is ey pass through, as shown by the letter A in Figure 2-3. A cessed more quickly if you use a pipeline because different fferent stages at the same time, as shown in Figure 2-4. A sophisticated than other processes: it can reroute data or s. Automobile assembly lines are good examples of pipe- ugh a pipeline and get a little work done at each step





with data flowing through it

Mixed Solutions

Consider the task of folding, stuffing, sealing, addressing, stamping, and mailing letters. If you assemble a group of six people for the task of stuffing many envelopes, you can arrange each person to specialize in and perform one task in a pipeline fashion (Figure 2-5). This contrasts with data parallelism, where you divide the supplies and give a batch of everything to each person (Figure 2-6). Each person then does all the steps on his collection of materials as his task.

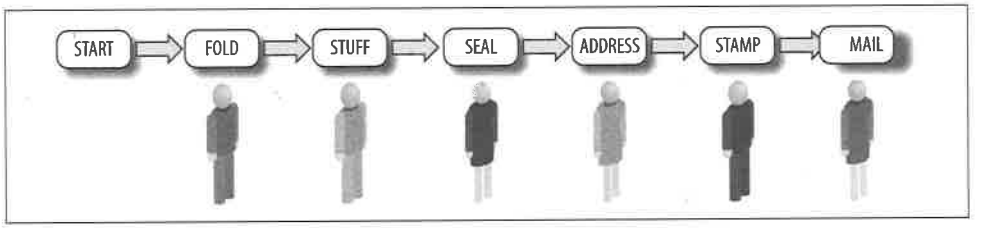


Figure 2-5. Pipelining—each person has a different job

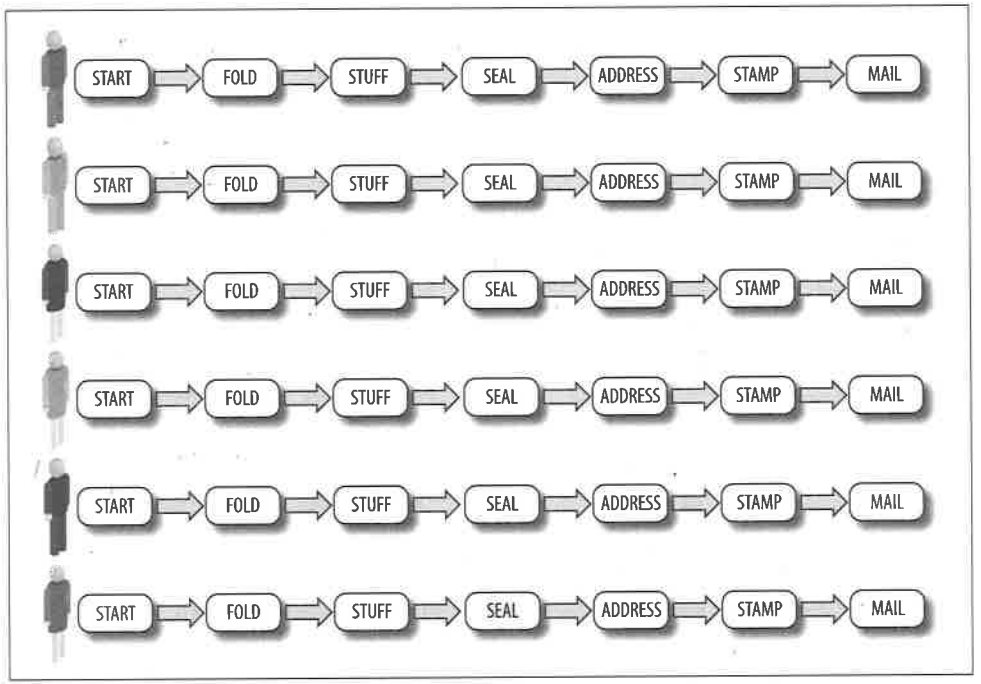


Figure 2-6. Data parallelism—each person has the same job

Figure 2-6 is clearly the right choice if every person has to work in a different location far from each other. That is called *coarse-grained* parallelism because the interactions between the tasks are infrequent (they only come together to collect envelopes, then leave and do their task, including mailing). The other choice shown

in Figure 2-5 is known as *fine-grained* parallelism because of the frequent interactions (every envelope is passed along to every worker in various steps of the operation).

Neither extreme tends to fit reality, although sometimes they may be close enough to be useful. In our example, it may turn out that addressing an envelope takes enough time to keep three people busy, whereas the first two steps and the last two steps require only one person on each pair of steps to keep up. Figure 2-7 illustrates the steps with the corresponding size of the work to be done. The resulting pipeline (Figure 2-8) is really a hybrid of data and task parallelism.

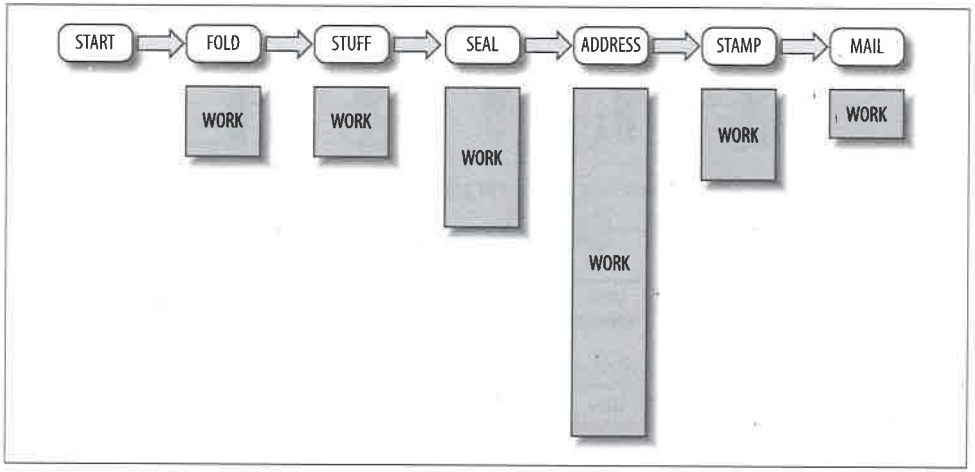


Figure 2-7. Unequal tasks are best combined or split to match people

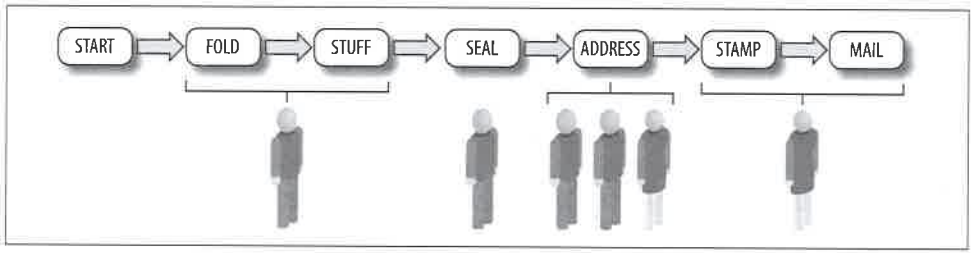


Figure 2-8. Because tasks are not equal, assign more people to the addressing task

Achieving Parallelism

Coordinating people around the job of preparing and mailing the envelopes is easily expressed by the following two conceptual steps:

1. Assign people to tasks (and feel free to move them around to balance the workload).
2. Start with one person on each of the six tasks, but be willing to split up a given task so that two or more people can work on it together.

The six tasks are folding, stuffing, sealing, addressing, stam
also have six people (resources) to help with the work
Threading Building Blocks works best: you define tasks and
explain and then split or combine data to match up with re
the work.

The first step in writing a parallel program is to consider w
Many textbooks wrestle with task and data parallelism as th
choice. Threading Building Blocks allows any combinatio
express.

If you are lucky, your program will be cleanly data-paralle
work, Threading Building Blocks requires you only to specif
them. For a completely data-parallel task, in Threading B
define one task to which you give all the data. That task wi
matically to use the available hardware parallelism. The impl
often eliminate the need for using locks to achieve synchroniz

People have been exploring decomposition for decades, a
emerged. We'll cover this more later when we discuss desi
programming.

Scaling and Speedup

The scalability of a program is a measure of how much *spee*
you add more and more processor cores. Speedup is the rati
run a program without parallelism versus the time it runs i
2X indicates that the parallel program runs in half the tim
gram. An example would be a sequential program that takes
one-processor machine and 17 seconds to run on a quad-core

As a goal, we would expect that our program running on two
run faster than the program running on one processor cor
four processor cores should be faster than running on two co

We say that a program does not *scale* beyond a certain p
processor cores no longer results in additional speedup. Wh
it is common for performance to fall if we force additiona
used. This is because the overhead of distributing and synch
nate. Threading Building Blocks has some algorithm templat
of a *grain size* to help limit the splitting of data to a reaso
problem. Grain size will be introduced and explained in detai

As Thinking Parallel becomes intuitive, structuring proble
second nature.

How Much Parallelism Is There in an Application?

The topic of how much parallelism there is in an application has gotten considerable debate, and the answer is "it depends."

It certainly depends on the size of the problem to be solved and on the ability to find a suitable algorithm to take advantage of the parallelism. Much of this debate previously has been centered on making sure we write efficient and worthy programs for expensive and rare parallel computers. The definition of size, the efficiency required, and the expense of the computer have all changed with the emergence of multi-core processors. We need to step back and be sure we review the ground we are standing on. The world has changed.

Amdahl's Law

Gene Amdahl, renowned computer architect, made observations regarding the maximum improvement to a computer system that can be expected when only a portion of the system is improved. His observations in 1967 have come to be known as *Amdahl's Law*. It tells us that if we speed up *everything* in a program by 2X, we can expect the resulting program to run 2X faster. However, if we improve the performance of only half the program by 2X, the overall system improves only by 1.33X. Amdahl's Law is easy to visualize. Imagine a program with five equal parts that runs in 500 seconds, as shown in Figure 2-9. If we can speed up two of the parts by 2X and 4X, as shown in Figure 2-10, the 500 seconds are reduced to only 400 and 350 seconds, respectively. More and more we are seeing the limitations of the portions that are not speeding up through parallelism. No matter how many processor cores are available, the serial portions create a barrier at 300 seconds that will not be broken (see Figure 2-11).

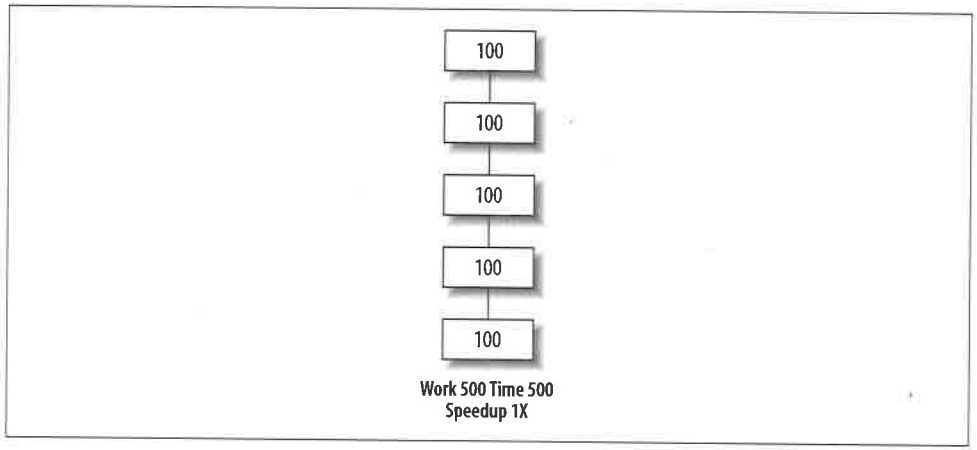


Figure 2-9. Original program without parallelism

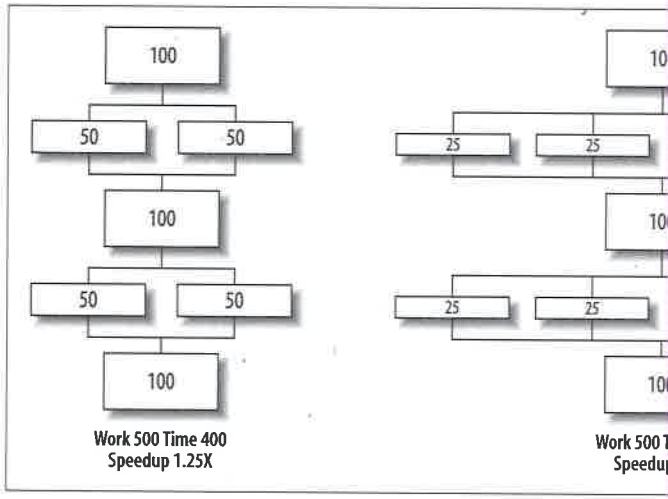


Figure 2-10. Progress on adding parallelism

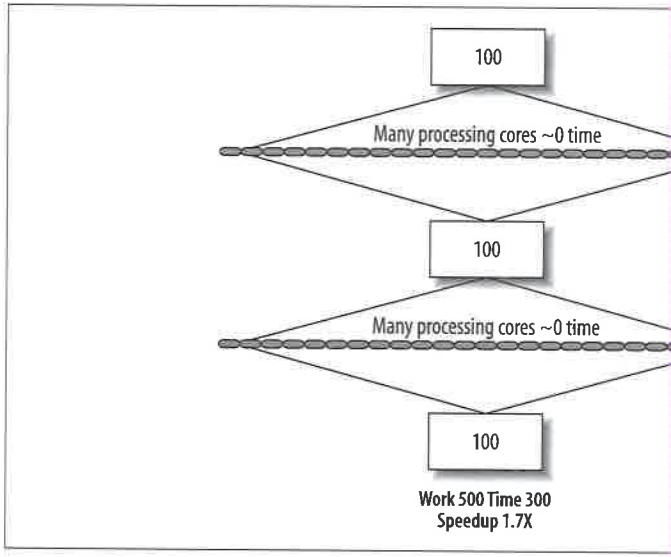


Figure 2-11. Limits according to Amdahl's Law

Parallel programmers have long used Amdahl's Law t
speedup that can be expected using multiple processors.
mately tells us that a computer program will never go faster
that do *not* run in parallel (the serial portions), no matter
have.

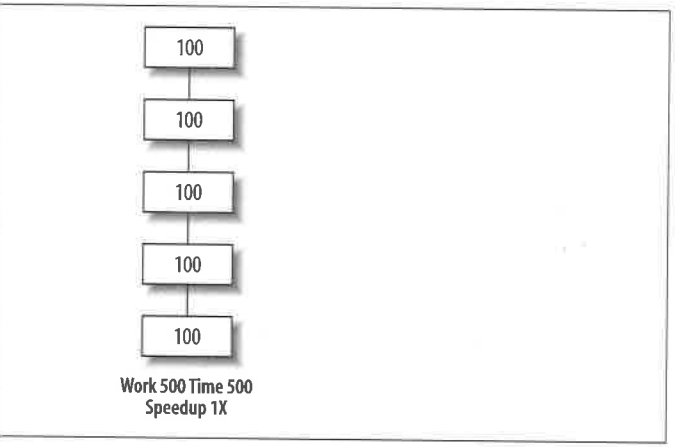
Many have used Amdahl's Law to predict doom and gloo
but there is another way to look at things that shows much

sm Is There in an Application?

parallelism there is in an application has gotten considerable
s "it depends."

he size of the problem to be solved and on the ability to find
ake advantage of the parallelism. Much of this debate previ-
on making sure we write efficient and worthy programs for
el computers. The definition of size, the efficiency required,
omputer have all changed with the emergence of multi-core
tep back and be sure we review the ground we are standing
ed.

computer architect, made observations regarding the maxi-
computer system that can be expected when only a portion
ed. His observations in 1967 have come to be known as
that if we speed up *everything* in a program by 2X, we can
gram to run 2X faster. However, if we improve the perfor-
rogram by 2X, the overall system improves only by 1.33X.
visualize. Imagine a program with five equal parts that runs
n in Figure 2-9. If we can speed up two of the parts by 2X
ure 2-10, the 500 seconds are reduced to only 400 and 350
ore and more we are seeing the limitations of the portions
through parallelism. No matter how many processor cores
portions create a barrier at 300 seconds that will not be



without parallelism

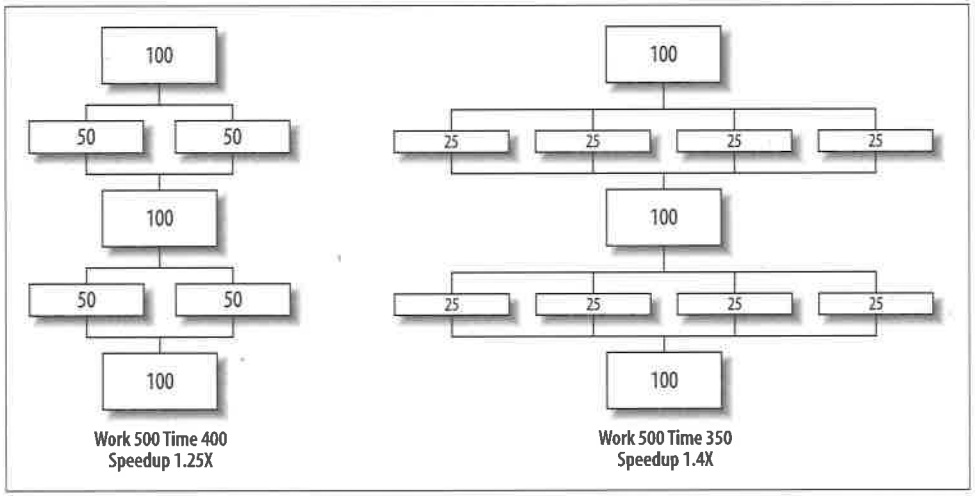


Figure 2-10. Progress on adding parallelism

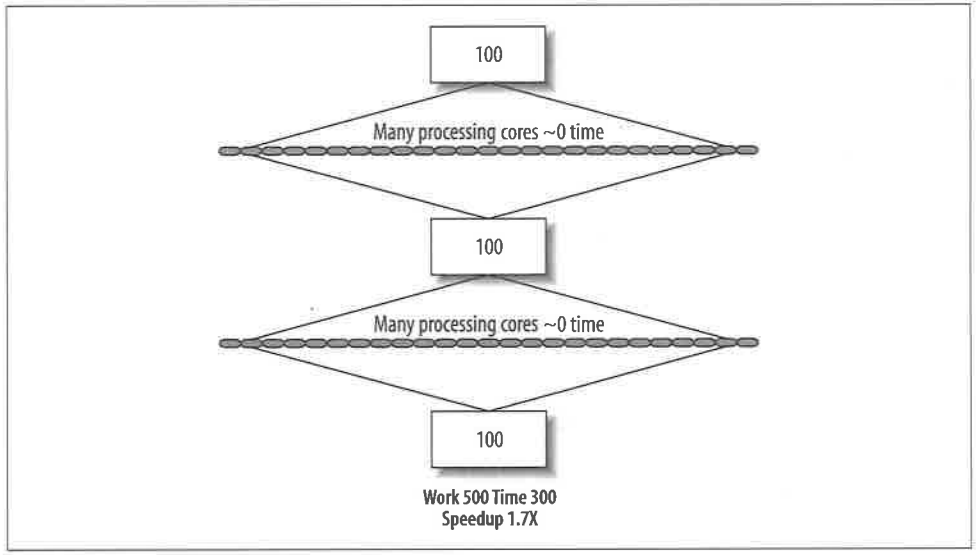


Figure 2-11. Limits according to Amdahl's Law

Parallel programmers have long used Amdahl's Law to predict the maximum speedup that can be expected using multiple processors. This interpretation ultimately tells us that a computer program will never go faster than the sum of the parts that do *not* run in parallel (the serial portions), no matter how many processors we have.

Many have used Amdahl's Law to predict doom and gloom for parallel computers, but there is another way to look at things that shows much more promise.

ograms as fixed, while we make changes to the computer. But dicate that as computers get new capabilities, applications ge of these features. Most of today's applications would not 10 years ago, and many would run poorly on machines that his observation is not limited to obvious applications such as o office applications, web browsers, photography software, iting software, and Google Earth.

s after the appearance of Amdahl's Law, John Gustafson, l Labs, took a different approach and suggested a reevalua- Gustafson noted that parallelism is more useful when you grow as computers become more powerful and support pro- k rather than focusing on a fixed workload. For many prob- ize grows, the work required for the parallel part of the an the part that cannot be parallelized (the so-called serial blem size grows, the serial fraction decreases and, according alability improves. So we can start with an application that : if the problem scales with the available parallelism, we are ments illustrated in Figure 2-12. If the sequential parts still time to perform, they become less and less important as a The algorithm eventually reaches the conclusion shown in grows at the same rate as the number of processors, which *n scaling*, denoted as O(*n*).

efficiency of the program is still greatly limited by the serial sing processors in our example is about 40 percent for large n a supercomputer, this might be a terrible waste. On a sys- cessors, one can hope that other work is running on the o use the processing power our application does not use. complexities. In any case, it is still good to minimize serial he "glass half empty" view and favor Amdahl's Law or you f full" view and favor Gustafson's observations.

Gustafson's observations are correct. The difference lies in e an existing program run faster with the same workload or a larger workload. History clearly favors programs getting g larger problems. Gustafson's observations fit the histori- heless, Amdahl's Law is still going to haunt us as we work plication work faster on the same benchmark. You have to ter picture.

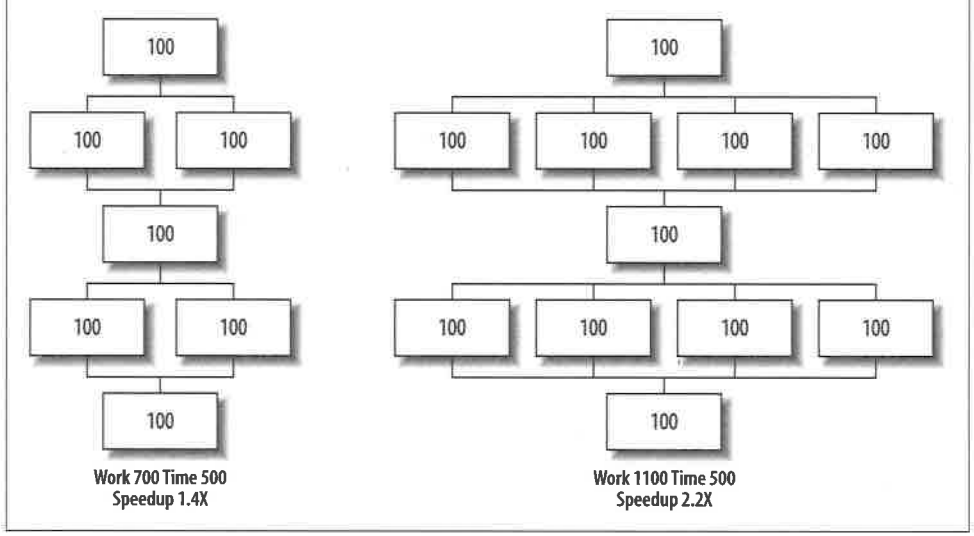


Figure 2-12. Scale the workload with the capabilities

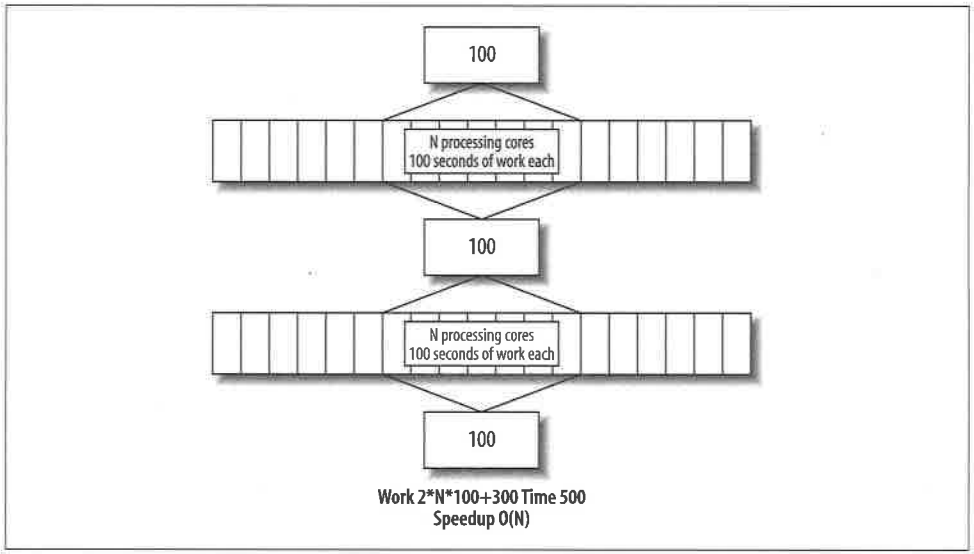


Figure 2-13. Gustafson saw a path to scaling

The value of parallelism is easier to prove if you are looking forward than if you assume the world is not changing.