Introduction to Scheduling

Arnaud Legrand, CNRS, University of Grenoble

LIG laboratory, arnaud.legrand@imag.fr

November 9, 2009
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
     - Adaptive Parallel Programs
     - Task Graphs and Parallel Tasks From Outer Space
2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation
3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?
4. Gang Scheduling as an Alternative
   - Principles
Matrix Product: Sequential Version

1 \{ To compute $C \leftarrow C + A \times B$ \}
2 \textbf{for} \ i = 1 \ \textbf{to} \ n \ \textbf{do}
3 \quad \textbf{for} \ j = 1 \ \textbf{to} \ n \ \textbf{do}
4 \quad \quad \textbf{for} \ k = 1 \ \textbf{to} \ n \ \textbf{do}
5 \quad \quad \quad \ C_{i,j} \leftarrow C_{i,j} + A_{i,k} \times B_{k,j}
Matrix Product: Sequential Version

\[ C_{1,1} \leftarrow C_{1,1} + A_{1,1} \times B_{1,1} \]

\[ C_{1,1} \leftarrow C_{1,1} + A_{1,2} \times B_{2,1} \]

\[ C_{1,2} \leftarrow C_{1,2} + A_{1,1} \times B_{1,2} \]

\[ C_{1,2} \leftarrow C_{1,2} + A_{1,2} \times B_{2,2} \]

\[ \ldots \]

\[ C_{1,1} \leftarrow C_{1,1} + A_{1,2} \times B_{2,1} \]

\[ A_{1,1} \quad A_{1,2} \]

\[ B_{1,1} \quad B_{1,2} \]

\[ A_{2,1} \quad A_{2,2} \]

\[ B_{2,1} \quad B_{2,2} \]

\[ C_{1,1} \quad C_{1,2} \]

\[ A_{1,1} \quad A_{1,2} \]

\[ C_{2,1} \quad C_{2,2} \]

\[ A_{2,1} \quad A_{2,2} \]
Matrix Product: Sequential Version

1. Load $C_{1,1}$, $A_{1,1}$, $B_{1,1}$
2. $C_{1,1} \leftarrow C_{1,1} + A_{1,1} \times B_{1,1}$
3. Unload $A_{1,1}$, $B_{1,1}$. Load $A_{1,2}$, $B_{2,1}$
4. $C_{1,1} \leftarrow C_{1,1} + A_{1,2} \times B_{2,1}$
5. Unload $C_{1,1}$, $A_{1,2}$, $B_{2,1}$. Load $C_{1,2}$, $A_{1,1}$, $B_{1,2}$
6. $C_{1,2} \leftarrow C_{1,2} + A_{1,1} \times B_{1,2}$
7. Unload $A_{1,1}$, $B_{1,2}$
8. $C_{1,2} \leftarrow C_{1,2} + A_{1,2} \times B_{2,2}$
9. ...

CPU

I/O

$B_{1,1}$ $B_{1,2}$

$B_{2,1}$ $B_{2,2}$

$A_{1,1}$ $A_{1,2}$

$C_{1,1}$ $C_{1,2}$

$A_{2,1}$ $A_{2,2}$

$C_{2,1}$ $C_{2,2}$
Matrix Product: Sequential Version

Sequential Programs

Sequential programs are generally a succession of CPU burst and I/O burst.
A, B, and C are initially located on the server.
We will distribute A, B, and C on P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}.
We will make use of all four processors to compute C \leftarrow C + A \times B.
Such a parallel program could be written using for example MPI. We want a SPMD algorithm.
Algorithm

1. \{ \text{P}_{i,j} \text{ is responsible for computing } C_{i,j}. \} \\
2. Load \( C_{i,j}, A_{(i+j)\%2}, B_{(i+j)\%2,j} \) from the server \\
3. \( C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \) \\
4. Exchange \( A_{\text{local}} \) with horizontal neighbor \\
5. Exchange \( B_{\text{local}} \) with vertical neighbor \\
6. \( C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \) \\
7. Unload \( C_{i,j} \) to the server
Matrix Product: Parallel Version (2/2)

Algorithm

1 \{ \textit{P}_{i,j} \text{ is responsible for computing } C_{i,j}. \} 
2 Load $C_{i,j}$, $A_{i,(i+j)\%2}$, $B_{(i+j)\%2,j}$ from the server 
3 $C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}}$
4 Exchange $A_{\text{local}}$ with horizontal neighbor 
5 Exchange $B_{\text{local}}$ with vertical neighbor 
6 $C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}}$
7 Unload $C_{i,j}$ to the server
Algorithm

\begin{itemize}
\item[1] \{ \( P_{i,j} \text{ is responsible for computing } C_{i,j} \). \}
\item[2] Load \( C_{i,j}, A_{i,(i+j)\%2}, B_{(i+j)\%2,j} \) from the server
\item[3] \( C_{local} \leftarrow C_{local} + A_{local} \times B_{local} \)
\item[4] Exchange \( A_{local} \) with horizontal neighbor
\item[5] Exchange \( B_{local} \) with vertical neighbor
\item[6] \( C_{local} \leftarrow C_{local} + A_{local} \times B_{local} \)
\item[7] Unload \( C_{i,j} \) to the server
\end{itemize}
Algorithm

1 \{ \text{\textit{P}}_{i,j} \text{ is responsible for computing } C_{i,j}. \} \\
2 \text{Load } C_{i,j}, A_{i,(i+j)\%2}, B_{(i+j)\%2,j} \text{ from the server} \\
3 C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \\
4 \text{Exchange } A_{\text{local}} \text{ with horizontal neighbor} \\
5 \text{Exchange } B_{\text{local}} \text{ with vertical neighbor} \\
6 C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \\
7 \text{Unload } C_{i,j} \text{ to the server}
Algorithm

1. \{ \textit{P} \textsubscript{i,j} is responsible for computing \textit{C} \textsubscript{i,j}. \}
2. Load \textit{C} \textsubscript{i,j}, \textit{A} \textsubscript{i,(i+j)\mod 2}, \textit{B} \textsubscript{(i+j)\mod 2, j} from the server
3. \textit{C} \textsubscript{local} ← \textit{C} \textsubscript{local} + \textit{A} \textsubscript{local} × \textit{B} \textsubscript{local}
4. Exchange \textit{A} \textsubscript{local} with horizontal neighbor
5. Exchange \textit{B} \textsubscript{local} with vertical neighbor
6. \textit{C} \textsubscript{local} ← \textit{C} \textsubscript{local} + \textit{A} \textsubscript{local} × \textit{B} \textsubscript{local}
7. Unload \textit{C} \textsubscript{i,j} to the server
Algorithm

1. \{ \( P_{i,j} \) is responsible for computing \( C_{i,j} \). \}
2. Load \( C_{i,j} \), \( A_{i,(i+j)\%2} \), \( B_{(i+j)\%2,j} \) from the server
3. \( C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \)
4. Exchange \( A_{\text{local}} \) with horizontal neighbor
5. Exchange \( B_{\text{local}} \) with vertical neighbor
6. \( C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \)
7. Unload \( C_{i,j} \) to the server
Matrix Product: Parallel Version (2/2)

Algorithm

\begin{itemize}
\item[1] \{ \( P_{i,j} \) is responsible for computing \( C_{i,j} \). \}
\item[2] Load \( C_{i,j}, A_{(i+j)\%2}, B_{(i+j)\%2,j} \) from the server
\item[3] \( C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \)
\item[4] Exchange \( A_{\text{local}} \) with horizontal neighbor
\item[5] Exchange \( B_{\text{local}} \) with vertical neighbor
\item[6] \( C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}} \)
\item[7] Unload \( C_{i,j} \) to the server
\end{itemize}
Algorithm

1 \{ \textit{P}_{i,j} \text{ is responsible for computing } C_{i,j}. \} 
2 Load $C_{i,j}$, $A_{i,(i+j)\%2}$, $B_{(i+j)\%2,j}$ from the server 
3 $C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}}$
4 Exchange $A_{\text{local}}$ with horizontal neighbor 
5 Exchange $B_{\text{local}}$ with vertical neighbor 
6 $C_{\text{local}} \leftarrow C_{\text{local}} + A_{\text{local}} \times B_{\text{local}}$
7 Unload $C_{i,j}$ to the server

Server

$P_{1,1}$

$P_{1,2}$

$P_{2,1}$

$P_{2,2}$
Matrix Product: Parallel Version (2/2)

Algorithm

1 \{ \textit{P}_{i,j} \text{ is responsible for computing } \textit{C}_{i,j}. \} \\
2 \text{Load } \textit{C}_{i,j}, \textit{A}_{i,(i+j)\%2}, \textit{B}_{(i+j)\%2,j} \text{ from the server} \\
3 \textit{C}_{\text{local}} \leftarrow \textit{C}_{\text{local}} + \textit{A}_{\text{local}} \times \textit{B}_{\text{local}} \\
4 \text{Exchange } \textit{A}_{\text{local}} \text{ with horizontal neighbor} \\
5 \text{Exchange } \textit{B}_{\text{local}} \text{ with vertical neighbor} \\
6 \textit{C}_{\text{local}} \leftarrow \textit{C}_{\text{local}} + \textit{A}_{\text{local}} \times \textit{B}_{\text{local}} \\
7 \text{Unload } \textit{C}_{i,j} \text{ to the server}

Parallel Programs

Parallel programs are generally a succession of CPU burst and communication burst. The synchronization pattern generally incurs idle time. This is the parallelization overhead.
Work, Cost, Speed-up and Efficiency

Server

$P_{1,1}$

$P_{1,2}$

$P_{2,1}$

$P_{2,2}$
**Definition: Work.**

The *work* is the *amount of computation* performed (the surface of the pink area).

In the previous parallel Matrix Multiplication example, the work is the same as in the sequential Matrix Multiplication example. However, parallel algorithms generally do not do the same operations as the sequential ones. They often have to do more. Therefore, the work $W(p)$ generally depends on the number of processors that are allotted!
Definition: **Cost**.

\[ C(p) = p \times \text{TotalTime}(p). \]

It is the total surface. The cost accounts for the idle time of the processing units.
### Definition: Speed-up and Efficiency

- **Speed-up:**  \[ s(p) = \frac{\text{SequentialTime}}{\text{TotalTime}(p)} \]

- **Efficiency:**  \[ e(p) = \frac{s(p)}{p} = \frac{\text{SequentialTime}}{p \times \text{TotalTime}(p)} \]
Side Note on Speed-up and Efficiency

**Speed-up**

We have $\text{SequentialTime} \leq C(p) \leq p \times \text{TotalTime}(p)$.

The speed-up is bounded by the number of processors and the efficiency is thus in $[0, 1]$.

Still, supralinear speed-up may happen!

We did not take I/O into account. With $p$ processors, we have $p$ times more available memory. Swapping sometimes kills the sequential algorithm.

Efficiency $\text{TotalTime}(p)$ does not necessarily decrease with $p$ due to the parallelization overhead.

Using more processors may hurt and may be particularly inefficient!
Side Note on Speed-up and Efficiency

**Speed-up**

We have \( \text{SequentialTime} \leq C(p) \leq p \times \text{TotalTime}(p) \).

Hence, \( s(p) = \frac{\text{SequentialTime}}{\text{TotalTime}(p)} \leq p \) and \( e(p) = \frac{\text{SequentialTime}}{p \times \text{TotalTime}(p)} \leq 1 \).

The speed-up is bounded by the number of processors and the efficiency is thus in \([0, 1]\).
Side Note on Speed-up and Efficiency

**Speed-up**

We have $\text{SequentialTime} \leq C(p) \leq p \times \text{TotalTime}(p)$.

Hence, $s(p) = \frac{\text{SequentialTime}}{\text{TotalTime}(p)} \leq p$ and $e(p) = \frac{\text{SequentialTime}}{p \times \text{TotalTime}(p)} \leq 1$.

The speed-up is bounded by the number of processors and the efficiency is thus in $[0, 1]$.

Still, **Supra-linear speed-up may happen!**
Side Note on Speed-up and Efficiency

**Speed-up**

We have \( \text{SequentialTime} \leq C(p) \leq p \times \text{TotalTime}(p) \).

Hence, \( s(p) = \frac{\text{SequentialTime}}{\text{TotalTime}(p)} \leq p \) and \( e(p) = \frac{\text{SequentialTime}}{p \times \text{TotalTime}(p)} \leq 1 \).

The speed-up is bounded by the number of processors and the efficiency is thus in \([0, 1]\).

Still, **Supra-linear speed-up may happen!**

We did not take I/O into account. With \( p \) processor, we have \( p \) times more available memory. Swapping sometimes kills the sequential algorithm.
Side Note on Speed-up and Efficiency

**Speed-up**

We have $\text{SequentialTime} \leq C(p) \leq p \times \text{TotalTime}(p)$.

Hence, $s(p) = \frac{\text{SequentialTime}}{\text{TotalTime}(p)} \leq p$ and $e(p) = \frac{\text{SequentialTime}}{p \times \text{TotalTime}(p)} \leq 1$.

The speed-up is bounded by the number of processors and the efficiency is thus in $[0, 1]$.

Still, **Supra-linear speed-up may happen!**

We did not take I/O into account. With $p$ processor, we have $p$ times more available memory. Swapping sometimes kills the sequential algorithm.

**Efficiency**

$\text{TotalTime}(p)$ does not necessarily decrease with $p$ due to the parallelization overhead.

Using more processors may hurt and may be particularly inefficient!
Parallel Matrix Algorithm
Block Version of the Outer-Product Algorithm

1 var $A$, $B$, $C$: array[0..$m-1$,0..$m-1$] of real
2 var buffer$A$, buffer$B$: array[0..$m-1$,0..$m-1$] of real
3 $q \leftarrow \text{SRQT(Num_PROCS())}$
4 $myrow \leftarrow \text{MY_PROC_Row()}$
5 $mycol \leftarrow \text{MY_PROC_Col()}$
6 for $k = 0$ to $q - 1$ do
7     for $i = 0$ to $m - 1$ do \{ Broadcast $A$ along rows \}
8     \quad $\text{BroadcastRow}(i, k, A, \text{buffer$A$}, m \times m)$
9     for $j = 0$ to $m - 1$ do \{ Broadcast $B$ along columns \}
10    \quad $\text{BroadcastCol}(k, j, B, \text{buffer$B$}, m \times m)$
11    \quad \{ Multiply matrix blocks \}
12    if ($myrow = k$) And ($mycol = k$) then
13        $\text{MatrixMultiplyAdd}(C, A, B, m)$
14    else if ($myrow = k$) then $\text{MatrixMultiplyAdd}(C, \text{buffer$A$}, B, m)$
15    else if ($mycol = k$) then $\text{MatrixMultiplyAdd}(C, A, \text{buffer$B$}, m)$
16    else $\text{MatrixMultiplyAdd}(C, \text{buffer$A$}, \text{buffer$B$}, m)$
var $A, B, C$: array[0..m−1,0..m−1] of real
var $bufferA, bufferB$: array[0..m−1,0..m−1] of real
$q ← \text{SRQT}(\text{NUM_PROCS}())$
$myrow ← \text{MY_PROC_ROW}()$

Two Comments

- Many parallel programs take the number of processors as an input and adapt to it.
- Many parallel programs use collective communication operations and synchronization.

{ Multiply matrix blocks }

if ($myrow = k$) And ($mycol = k$) then
  $\text{MATRIXMULTIPLYADD}(C, A, B, m)$
else if ($myrow = k$) then
  $\text{MATRIXMULTIPLYADD}(C, bufferA, B, m)$
else if ($mycol = k$) then
  $\text{MATRIXMULTIPLYADD}(C, A, bufferB, m)$
else
  $\text{MATRIXMULTIPLYADD}(C, bufferA, bufferB, m)$
**Bulk Synchronous Parallel** is a programming paradigm whose principle is a series of independent steps of computations and communication/synchronization.

The cost of a superstep is determined as the sum of three terms:

\[ T = \max_i w(i) + \max h(i)g + l \]

Scheduling under BSP is about finding a tradeoff between load-balancing and number of communication/synchronizations.
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
     - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Remember the previous “Block Version of the Outer-Product Algorithm”.

```
1  q ← √(NUM_PROCS())
2  myrow ← MY_PROC_ROW()
3  mycol ← MY_PROC_COL()
4  for k = 0 to q - 1 do
5     ...
```

This $q$ is not hard-coded. The algorithm adapts to the number of available processors at the beginning of the execution. It uses this number to distribute the data and organize the communications.

Such programs are called moldable.
When using adaptive mesh-refining, load imbalance occurs. Coupling code makes it worse. Recomputing a good partition and redistributing the data is not necessarily a good option. However, adding computing resources on the fly is often very efficient. The resource requirements vary over time. This kind of program is called dynamic.
When using adaptive mesh-refining, load imbalance occurs. Coupling code makes it worse. Recomputing a good partition and redistributing the data is not necessarily a good option. However, adding computing resources on the fly is often very efficient. The resource requirements vary over time. This kind of program is called dynamic.

A. Legrand (CNRS-LIG) INRIA-MESCAL
When using adaptive mesh-refining, load imbalance occurs. Coupling code makes it worse.
When using adaptive mesh-refining, **load imbalance** occurs. Coupling code makes it worse. Recomputing a good partition and redistributing the data is not necessarily a good option. However, **adding computing resources** on the fly is often very efficient.

The resource requirements vary over the time.

This kind of program is called **dynamic**.
These programs can adapt to the resource they are allotted over the time.
This kind of program is called malleable.
KAAPPI: Adaptive, Asynchronous Parallel and Interactive Computing

KAAPPI is based on work-stealing algorithms and contains non-blocking and scalable algorithms.
KAAPPI/Taktuk won the 4th and 5th International Challenge GRIDS@WORK (2007, 2008).

2007 N-queens

2008 Super Quant Monte-Carlo, pricing application.
- 3609 cores used between France and Japan during one hour.
- The KAAPPI/Taktuk team was able to price 988 actions on the 1000 of the challenge and was scored 8760/18000.
- The second team was able to price 177 actions using 4329 and was scored 1459/18000.

These programs can adapt to the resource they are allotted over the time.
This kind of program is called malleable.
Parallelizing generally has a price. There is a computation overhead and a communication/synchronization overhead. Some applications however have a very low computation overhead and can be very easily divided.

- Pattern Searching
- Database Computation
- Video encoding
- Image processing

Such applications are very well suited to master-slave computing.

Computation times and communication times are linear with the fraction of allotted load.

This kind of program is called divisible.
Parallelizing generally has a price. There is a computation overhead and a communication/synchronization overhead. Some applications however have a very low computation overhead and can be very easily divided.

- Pattern Searching
- Database Computation
- Video encoding
- Image processing

Such applications are very well suited to master-slave computing.

\[
\begin{array}{ccc}
\alpha_1 g & \alpha_2 g & \alpha_p g \\
T_1 & T_2 & T_p & T_f \\
\end{array}
\]

Computation times and communication times are linear with the fraction of allotted load.

This kind of program is called divisible.
Parallelizing generally has a price. There is a computation overhead and a communication/synchronization overhead. Some applications however have a very low computation overhead and can be very easily divided.

- Pattern Searching
- Database Computation
- Video encoding
- Image processing

Such applications are very well suited to master-slave computing.

Computation times and communication times are linear with the fraction of allotted load.

This kind of program is called divisible.
Parallelizing generally has a price. There is a computation overhead and a communication/synchronization overhead. Some applications however have a very low computation overhead and can be very easily divided.

- Pattern Searching
- Database Computation
- Video encoding
- Image processing

Such applications are very well suited to master-slave computing.

Computation times and communication times are linear with the fraction of allotted load.

This kind of program is called divisible.
Parallelizing generally has a price. There is a computation overhead and a communication/synchronization overhead. Some applications however have a very low computation overhead and can be very easily divided.

- Pattern Searching
- Database Computation
- Video encoding
- Image processing

Such applications are very well suited to master-slave computing.

Computation times and communication times are linear with the fraction of allotted load.

This kind of program is called divisible.
Outline

1 Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2 Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3 Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4 Gang Scheduling as an Alternative
   - Principles
Solving $A.x = B$ where $A$ is lower triangular matrix

for $i = 1$ to $n$ do

- Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$

for $j = i + 1$ to $n$ do

- Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$

$A.X = B$
Analyzing a Simple Code

Solving $A \cdot x = B$ where $A$ is lower triangular matrix

for $i = 1$ to $n$ do

Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$

for $j = i + 1$ to $n$ do

Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$

For a given value $1 \leq i \leq n$, all tasks $T_{i,*}$ are computations done during the $i^{th}$ iteration of the outer loop.
Analyzing a Simple Code

Solving $A.x = B$ where $A$ is lower triangular matrix

for $i = 1$ to $n$ do

<table>
<thead>
<tr>
<th>Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $j = i + 1$ to $n$ do</td>
</tr>
<tr>
<td>Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$</td>
</tr>
</tbody>
</table>

For a given value $1 \leq i \leq n$, all tasks $T_{i,*}$ are computations done during the $i^{th}$ iteration of the outer loop.

$<_{seq}$ is the sequential order:

$$T_{1,1} <_{seq} T_{1,2} <_{seq} T_{1,3} <_{seq} \cdots <_{seq} T_{1,n} <_{seq} T_{2,2} <_{seq} T_{2,3} <_{seq} \cdots <_{seq} T_{n,n}.$$
However, some independent tasks could be executed in parallel. Independent tasks are the ones whose execution order can be changed without modifying the result of the program. Two independent tasks may read the value but never write to the same memory location.
However, some independent tasks could be executed in parallel. Independent tasks are the ones whose execution order can be changed without modifying the result of the program. Two independent tasks may read the value but never write to the same memory location.

For a given task $T$, $\text{In}(T)$ denotes the set of input variables and $\text{Out}(T)$ the set of output variables.
However, some independent tasks could be executed in parallel. Independent tasks are the ones whose execution order can be changed without modifying the result of the program. Two independent tasks may read the value but never write to the same memory location.

For a given task $T$, $\text{In}(T)$ denotes the set of input variables and $\text{Out}(T)$ the set of output variables. In the previous example, we have:

$$
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do } & \quad \text{Task } T_{i,i}: \quad x(i) \leftarrow b(i)/a(i,i) \\
\text{for } j = i + 1 \text{ to } n \text{ do } & \quad \text{Task } T_{i,j}: \quad b(j) \leftarrow b(j) - a(j,i) \times x(i)
\end{align*}
$$

\begin{align*}
\text{In}(T_{i,i}) &= \{b(i), a(i,i)\} \\
\text{Out}(T_{i,i}) &= \{x(i)\} \quad \text{and} \\
\text{In}(T_{i,j}) &= \{b(j), a(j,i), x(i)\} \\
\text{Out}(T_{i,j}) &= \{b(j)\} \text{ for } j > i.
\end{align*}
Two tasks \( T \) and \( T' \) are not independent (\( T \perp T' \)) whenever they share a written variable:

\[
T \perp T' \iff \begin{cases} 
  \text{In}(T) \cap \text{Out}(T') \neq \emptyset \\
  \text{or } \text{Out}(T) \cap \text{In}(T') \neq \emptyset \\
  \text{or } \text{Out}(T) \cap \text{Out}(T') \neq \emptyset
\end{cases}
\]

Those conditions are known as Bernstein’s conditions [Bernstein66].
Bernstein Conditions

Definition.

Two tasks $T$ and $T'$ are not independent ($T \perp T'$) whenever they share a written variable:

\[ T \perp T' \iff \begin{cases} \text{In}(T) \cap \text{Out}(T') \neq \emptyset \\ \text{or} \quad \text{Out}(T) \cap \text{In}(T') \neq \emptyset \\ \text{or} \quad \text{Out}(T) \cap \text{Out}(T') \neq \emptyset \end{cases}. \]

Those conditions are known as Bernstein’s conditions [Bernstein66].

We can check that:

\[
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do} & \\
\quad \text{Task } T_{i,i}: & x(i) \leftarrow b(i)/a(i,i) \\
\text{for } j = i+1 \text{ to } n \text{ do} & \\
\quad \text{Task } T_{i,j}: & b(j) \leftarrow b(j) - a(j,i) \times x(i)
\end{align*}
\]
Bernstein Conditions

Definition.

Two tasks $T$ and $T'$ are not independent ($T \perp T'$) whenever they share a written variable:

$$T \perp T' \iff \begin{cases} \text{In}(T) \cap \text{Out}(T') \neq \emptyset \\ \text{or} \quad \text{Out}(T) \cap \text{In}(T') \neq \emptyset \\ \text{or} \quad \text{Out}(T) \cap \text{Out}(T') \neq \emptyset \end{cases}.$$ 

Those conditions are known as Bernstein’s conditions [Bernstein66].

We can check that:

- $\text{Out}(T_{1,1}) \cap \text{In}(T_{1,2}) = \{x(1)\}$ for $i = 1$ to $n$ do
  - Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$
  - for $j = i + 1$ to $n$ do
    - Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$

  $T_{1,1} \perp T_{1,2}$. 

A. Legrand (CNRS-LIG) INRIA-MESCAL  Introduction to Scheduling  Applications 20 / 80
Bernstein Conditions

**Definition.**

Two tasks $T$ and $T'$ are not independent ($T \perp T'$) whenever they share a written variable:

$$T \perp T' \iff \begin{cases} 
\text{In}(T) \cap \text{Out}(T') \neq \emptyset \\
\text{or } \text{Out}(T) \cap \text{In}(T') \neq \emptyset \\
\text{or } \text{Out}(T) \cap \text{Out}(T') \neq \emptyset 
\end{cases}$$

Those conditions are known as Bernstein’s conditions [Bernstein66].

We can check that:

- $\text{Out}(T_{1,1}) \cap \text{In}(T_{1,2}) = \{x(1)\}$
  $$\leadsto T_{1,1} \perp T_{1,2}.$$  

- $\text{Out}(T_{1,3}) \cap \text{Out}(T_{2,3}) = \{b(3)\}$
  $$\leadsto T_{1,3} \perp T_{2,3}.$$
If $T \perp T'$, then they should be ordered with the sequential execution order. $T \prec T'$ if $T \perp T'$ and $T <_{\text{seq}} T'$.

More precisely, $\prec$ is defined as the **transitive closure** of $(<_{\text{seq}} \cap \perp)$. 

This implies that $T_1 \prec T_2$ means that $T_1$ must be finished before $T_2$ can start.
If $T \perp T'$, then they should be ordered with the sequential execution order. $T \prec T'$ if $T \perp T'$ and $T <_{seq} T'$.

More precisely $\prec$ is defined as the transitive closure of $(<_{seq} \cap \perp)$.

\[
\begin{align*}
\text{for } i &= 1 \text{ to } n \text{ do} \\
&\quad \text{Task } T_{i,i} : x(i) \leftarrow b(i) / a(i,i) \\
&\quad \text{for } j = i + 1 \text{ to } n \text{ do} \\
&\quad &\quad \text{Task } T_{i,j} : b(j) \leftarrow b(j) - a(j,i) \times x(i)
\end{align*}
\]

A dependence graph $G$ is used.
If $T \perp T'$, then they should be ordered with the sequential execution order. $T \prec T'$ if $T \perp T'$ and $T <_{seq} T'$.

More precisely $\prec$ is defined as the transitive closure of ($<_{seq} \cap \perp$).

```
for i = 1 to n do
    Task $T_{i,i}$: $x(i) \leftarrow b(i)/a(i,i)$
    for j = i + 1 to n do
        Task $T_{i,j}$: $b(j) \leftarrow b(j) - a(j,i) \times x(i)$
```

A dependence graph $G$ is used.

$(e : T \rightarrow T') \in G$ means that $T'$ can start only if $T$ has already been finished. $T$ is a predecessor of $T'$. 
If $T \perp T'$, then they should be ordered with the sequential execution order. $T \prec T'$ if $T \perp T'$ and $T \prec_{seq} T'$.

More precisely $\prec$ is defined as the transitive closure of $(\prec_{seq} \cap \perp)$.

\[
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do} \\
&\quad \text{Task } T_{i,i} : x(i) \leftarrow b(i)/a(i,i) \\
&\quad \text{for } j = i + 1 \text{ to } n \text{ do} \\
&\quad\quad \text{Task } T_{i,j} : b(j) \leftarrow b(j) - a(j,i) \times x(i)
\end{align*}
\]

A dependence graph $G$ is used.

$(e : T \rightarrow T') \in G$ means that $T'$ can start only if $T$ has already been finished. $T$ is a predecessor of $T'$.

Transitivity arcs are generally omitted.
The previous task graph comes from a low-level analysis of the code. It probably makes little sense to do a parallel implementation with MPI with such a low task granularity. Can totally make sense with OpenMP. Such task graphs can also be used by compilers to do code optimization by exploiting multiple functional units, pipelines functional units, etc. With blocking these tasks could become MPI (parallel) tasks.
The previous task graph comes from a **low-level** analysis of the code. It probably makes little sense to do a parallel implementation with **MPI** with such a low task granularity. Can totally make sense with **OpenMP**. Such task graphs can also be used by compilers to do code optimization by exploiting multiple functional units, pipelines functional units, etc. With **blocking** these tasks could become MPI (parallel) tasks.
Hide applications’ complexity
... to Parallel Tasks

Hide applications’ complexity
... to Parallel Tasks

Hide applications’ complexity
... to Parallel Tasks

Hide applications’ complexity
Hide applications’ complexity

3 versions:
  ▶ Rigid Tasks

The execution time \textit{generally} decreases with the number of processors but the penalty incurred by communications and synchronizations increases.
to Parallel Tasks

Hide applications’ complexity
3 versions:
- Rigid Tasks
- Moldable Tasks

The execution time *generally* decreases with the number of processors but the penalty incurred by communications and synchronizations increases.
Hide applications’ complexity

3 versions:
- Rigid Tasks
- Moldable Tasks
- Malleable Tasks

The execution time *generally* decreases with the number of processors but the penalty incurred by communications and synchronizations increases.
Task-graph do not necessarily come from instruction-level analysis.

```sql
select p.proteinID, 
    blast(p.sequence) 
from proteins p, proteinTerms t 
where p.proteinID = t.proteinID and 
t.term = GO:0008372
```
Task-graph do not necessarily come from instruction-level analysis.

\[
\text{select } p.\text{proteinID}, \\
\text{blast}(p.\text{sequence}) \\
\text{from proteins } p, \text{ proteinTerms } t \\
\text{where } p.\text{proteinID} = t.\text{proteinID} \text{ and } \\
t.\text{term} = \text{GO:0008372}
\]

▶ Each task may be a parallel job...
Task-graph do not necessarily come from instruction-level analysis.

select p.proteinID, 
    blast(p.sequence) 
from proteins p, proteinTerms t 
where p.proteinID = t.proteinID and 
t.term = GO:0008372

Each task may be a parallel job...
Each edge depicts a dependency i.e. most of the times some data to transfer.
I have presented you a few different parallel program models:

- rigid jobs
- moldable jobs
- dynamic jobs
- malleable jobs
- divisible jobs
- BSP jobs
- DAGs of the previous jobs

The rationale behind all these models is:

- the diversity and the complexity of parallel programs;
- the level of details we need/wish to expose to the one in charge of the execution.

**Modeling is an art.**

You have to know your application to know what is negligible and what is important. Even if your model is imperfect, you may still derive interesting results.
Outline

1 Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2 Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3 Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4 Gang Scheduling as an Alternative
   - Principles
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Some questions you need to answer

**Preemption** Are we allowed to suspend a program and resume it later?
- Resumed from the beginning or from where it was stopped?
- May be resumed on another machine or not (migration)?
- Does preemption/migration has a cost or not?

**Release dates** Are all tasks available at the very beginning or not?

**Deadlines** Are the tasks associated to a deadline before which they should complete? What happens when the deadline is missed?

**Dependencies** Are there dependencies between tasks (DAGs)?

**Users** Are there many users and should this be taken into account?

**Long-term vs. short-term** What kind of constraints do you have on the time needed to take your scheduling decisions?
What kind of information do you have to make your scheduling choices?

**Off-line** You know everything (release dates and processing time of each task) at the very beginning. It is the “simplest” setting and will give you insights on your scheduling problem even though these hypothesis do not really hold in practice. This kind of problem should thus be studied *before everything else*.

**On-line/clairvoyant** You do not know in advance when tasks arrive. However, once a new task are available, you know its computation time.

**On-line/non clairvoyant** You know nothing!

Sometimes (often?), reality is in between:

- We could have “informations” about the task arrival (e.g., periodic creation, random process, use the past to predict the future).
- We could have “informations” about the task computation requirement (e.g., mix of short tasks and long tasks).
Outline

1 Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2 Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3 Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4 Gang Scheduling as an Alternative
   - Principles
Criteria: Intuitive Notion

**CPU utilization** (max) percent usage of CPU. Only useful computations (mix CPU, I/O; preemption overhead).

**Throughput** (max) *average* number of tasks that complete their execution per time-unit.

**Makespan** (min) Completion time of the last finishing task.

**Load** (min) Completion time of the last finishing task for a given processor.

**Turnaround Time/Response Time/Flow** (min) amount of time it takes between the task arrival and its completion.

**Waiting Time** (min) amount of time spent waiting for being executed.

**Slowdown/Stretch** (min) slowdown factor encountered by a task relative to the time it would take on an unloaded system.

The previous quantities are task- or CPU-centric and need to be aggregated into a single objective function.

- **max** (the worst case)
- **average**: arithmetic (i.e. sum) or something else...
- **variance** (to be “fair” between the tasks).
A given task $T_i$ is defined by:

- processing time $p_i$
- release date $r_i$
- completion time $C_i$
- (number of required processors $q_i$)
- (deadline $d_i$)

### Completion Time

- **Makespan:** $C_{\text{max}} = \max_i C_i$
  This metric is the most classical and is relevant when scheduling a *single* application.
- **Total Completion Time:** $SC = \sum_i C_i$
Criteria: Classical Definitions

A given task $T_i$ is defined by:
- processing time $p_i$
- release date $r_i$
- completion time $C_i$
- (number of required processors $q_i$)
- (deadline $d_i$)

Response Time

$$F_i = C_i - r_i$$

- Maximum Flow Time: $F_{\text{max}} = \max_i F_i$
- Total Completion Time: $SF = \sum_i F_i = SC - \sum_i r_i$
A given task $T_i$ is defined by:

- processing time $p_i$
- release date $r_i$
- completion time $C_i$
- (number of required processors $q_i$)
- (deadline $d_i$)

**Waiting Time**

$$W_i = C_i - r_i - p_i$$

- Maximum Waiting time: $W_{\text{max}} = \max_i W_i$
- Total Waiting Time: $SW = \sum_i W_i = SF - \sum_i p_i$
A given task $T_i$ is defined by:

- processing time $p_i$
- release date $r_i$
- completion time $C_i$
- (number of required processors $q_i$)
- (deadline $d_i$)

**Slowdown**

$$S_i = \frac{C_i - r_i}{p_i}$$

- Maximum Stretch: $S_{\text{max}} = \max_i S_i$
- Total Stretch: $SS = \sum_i S_i$
1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Most scheduling problems are NP-complete but you may be lucky... So the first question to answer is: P or NP-hard?

For a given objective function $Obj$:

**Definition: Decision and Optimization.**

- $\text{Dec}(M)$: Is there a schedule $\sigma$ such that $Obj(\sigma) \leq M$?

- $\text{Opt}$: Find $M^*$ such that $M^* = \min_{\sigma} Obj(\sigma)$.

If $\text{Dec}$ can be solved in polynomial time, then so can $\text{Opt}$ (using a dichotomy). And conversely...

Note that since $SW(\sigma) = SF(\sigma) - \sum_i p_i = SC(\sigma) - \sum_i r_i - \sum_i p_i$, all these problems are equivalent on a complexity point of view.
Worst-Case Analysis: $\varrho$-approximation

Your scheduling problem is NP-hard so you need to propose a heuristic and compare it to the best possible solution. Consider a given objective function $Obj$.

**Definition: $\varrho$-approximation.**

An algorithm $A$ is a $\varrho$-approximation iff for any instance $I$, $Obj(A(I)) \leq \varrho \cdot Obj^*(I)$.

The approximation ratio of $A$ is:

$$\varrho(A) = \max_I \frac{Obj(A(I))}{Obj^*(I)}$$

Note that even though $SW(\sigma) = SF(\sigma) - \sum_i p_i = SC(\sigma) - \sum_i r_i - \sum_i p_i$, these problem are **not** equivalent on an approximation point of view.
Worst-Case Analysis: $\varrho$-competitive

What is the best solution to an online problem (where the heuristic doesn’t know in advance the jobs arrival)? We keep comparing to the best possible solution, i.e. the one that knows everything.

**Definition: $\varrho$-competitive.**

An algorithm $\mathcal{A}$ is a $\varrho$-approximation iff

for any instance $I$, $\text{Obj}(\mathcal{A}(I)) \leq \varrho \cdot \text{Obj}^*(I)$.

The approximation ratio of $\mathcal{A}$ is:

$$\varrho(\mathcal{A}) = \max I \frac{\text{Obj}(\mathcal{A}(I))}{\text{Obj}^*(I)}$$

It is the same definition except that it applies to online algorithms. For such a pessimistic evaluation, one commonly uses an adversary.
Average-Case Analysis

If we have a probability distribution over the set of instances, $Obj$ can thus be seen as a random variable. We can define the expectation of $Obj$.

$$
\mathbb{E}[Obj(A)] = \int_I Obj(A(I))p(I).dI = \sum_I Obj(A(I))p(I)
$$

People often try to evaluate at (at least through experiments)

$$
\varrho(A) = \int_I \frac{Obj(A(I))}{Obj^*(I)}p(I).dI \quad \text{or}
$$

$$
\varrho(A) = \frac{\mathbb{E}[Obj(A)]}{\mathbb{E}[Obj^*]} = \frac{\int_I Obj(A(I))p(I).dI}{\int_I Obj^*(I)p(I).dI}
$$

However, in the literature, there are many different ways of comparing random variables (and thus to compare and evaluate algorithms). These techniques will be presented in much more details later.
Outline

1 Modeling Applications, General Notions
   • Introducing Fundamental Notions Through the Matrix Product Example
   • Adaptive Parallel Programs
   • Task Graphs and Parallel Tasks From Outer Space

2 Defining a Scheduling Problem
   • Rules of the Game
   • Criteria: How Do You Win the Game?
   • Analysis Method
   • Graham Notation

3 Batch Scheduling
   • Principles
   • Theoretical results
   • Basic idea: FCFS + Backfilling
   • EASY
   • How Good is the Schedule?

4 Gang Scheduling as an Alternative
   • Principles
Many parameter can change in a scheduling problem. Graham has then proposed the following classification: $\langle \alpha | \beta | \gamma \rangle$ [Brucker-Book]
Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification: \( \langle \alpha | \beta | \gamma \rangle \) [Brucker-Book]

- \( \alpha \) is the processor environment (a few examples):
  - \( \emptyset \): single processor;
  - \( P \): identical processors;
  - \( Q \): uniform processors;
  - \( R \): unrelated processors;

- \( \beta \) describe task and resource characteristics (a few examples):
  - \( \text{pmtn} \): preemption;
  - \( \text{prec, tree or chains} \): general precedence constraints, tree constraints, set of chain constraints and independent tasks otherwise;
  - \( \tilde{d} \): deadlines;

- \( \gamma \) denotes the optimization criterion (a few examples):
  - \( C_{\text{max}} \): makespan;
  - \( \sum C_i \): average completion time;
  - \( \sum w_i C_i \): weighted A.C.T;
  - \( L_{\text{max}} \): maximum lateness \( (\max C_i - d_i) \);

...
Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification: \( \langle \alpha | \beta | \gamma \rangle \) [Brucker-Book]

- **\( \alpha \)** is the processor environment (a few examples):
  - \( \emptyset \): single processor;
  - \( P \): identical processors;
  - \( Q \): uniform processors;
  - \( R \): unrelated processors;

- **\( \beta \)** describe task and resource characteristics (a few examples):
  - \( pmtn \): preemption;
  - \( prec, \ tree \ or \ chains \): general precedence constraints, tree constraints, set of chain constraints and independent tasks otherwise;
  - \( r_j \): tasks have release dates;
  - \( p_j = p \) or \( p \leq p_j \leq \bar{p} \): all task have processing time equal to \( p \), or comprised between \( p \) and \( \bar{p} \), or have arbitrary processing times otherwise;
  - \( \bar{d} \): deadlines;
Many parameter can change in a scheduling problem. Graham has then proposed the following classification: $\langle \alpha | \beta | \gamma \rangle$ [Brucker-Book]

- $\alpha$ is the processor environment (a few examples):
  - $\emptyset$: single processor;
  - $P$: identical processors;
  - $Q$: uniform processors;
  - $R$: unrelated processors;

- $\beta$ describe task and resource characteristics (a few examples):
  - $pmtn$: preemption;
  - $prec$, $tree$ or $chains$: general precedence constraints, tree constraints, set of chain constraints and independent tasks otherwise;
  - $r_j$: tasks have release dates;

- $\gamma$ denotes the optimization criterion (a few examples):
  - $C_{\text{max}}$: makespan;
  - $\sum C_i$: average completion time;
  - $\sum w_i C_i$: weighted A.C.T;
  - $L_{\text{max}}$: maximum lateness ($\max C_i - d_i$);
  - $\ldots$
Understand the following problems and propose a practical situation to illustrate them:

- $\langle P|prec|C_{\text{max}} \rangle$
- $\langle P|q_j, prec|C_{\text{max}} \rangle$
- $\langle P|q_j|F_{\text{max}} \rangle$
- $\langle 1|r_j; pmtn|S_{\text{max}} \rangle$
- $\langle 1|r_j; pmtn, d_i|L_{\text{max}} \rangle$
Scheduling is a very generic word that encompass a very wide range of situations, problems and analysis techniques.

Scheduling is generally about deciding *who*, *where* and *when*.

It is thus almost everywhere and when you start looking at a given scheduling problem, with very high probability, many people already worked on it.

Doing a serious and **thorough bibliographical study** is thus of uttermost importance!
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

When one purchases a cluster, typically many users want to use it. One cannot let them step on each other's toes. Every user wants to be on a dedicated machine. Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc. The Job Scheduler is the entity that prevents them from stepping on each other's toes. The Job Scheduler gives out nodes to applications.
Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

A graph showing the relationship between time and processors with a rectangular area representing the time period during which a set of processors are busy.
Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other’s toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.
Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other’s toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

The Job Scheduler is the entity that prevents them from stepping on each other's toes.

The Job Scheduler gives out nodes to applications.
Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

When one purchases a cluster, typically many users want to use it.
- One cannot let them step on each other’s toes
- Every user wants to be on a dedicated machine
- Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.
Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other’s toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

The Job Scheduler is the entity that prevents them from stepping on each other’s toes

The Job Scheduler gives out nodes to applications
Each job is defined as a **Number of nodes** \((q_i)\) and a **Time** \((p_i)\):

\[
I \text{ want 6 nodes for 1h}
\]

Typically users are “charged” against an “allocation”: e.g. “You only get 100 CPU hours per week”.

A batch scheduler is a central middleware to manage resources (e.g. processors) of parallel machines:

- accept jobs (computing tasks) submitted by users
- decide **when** and **where** jobs are executed
- start jobs execution

They take into account:

- **unavailability** of some nodes
- users jobs **mutual exclusion**
- **specific needs** for jobs (memory, network, ...)

While trying to:

- **maximize resources usage**
- be **fair** among users
Batch Scheduling

Typical wanted features:

- Interactive mode
- Batch mode
- Parallel jobs support
- Multi-queues with priorities
- Admission policies (limit on usage, notions of user groups, power users)
- Resources matching
- File staging
- Jobs dependences
- Backfilling
- Reservations
- Best effort jobs
- Environment reconfiguration

There are many existing batch schedulers: LSF, PBS/Torque, Maui scheduler, Sun Grid Engine, EASY, OAR, ...

These are complex systems with many config options!
## Main Batch Schedulers Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>OpenPBS</th>
<th>SGE</th>
<th>Maui Scheduler (+ OpenPBS)</th>
<th>OAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interactive mode</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Batch mode</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Parallel jobs support</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Multi-queues with priorities</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Resources matching</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Admission policies</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>File staging</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Jobs dependences</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Backfilling</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reservations</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best effort jobs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Environment reconfiguration</td>
<td></td>
<td></td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Fair sharing</td>
<td></td>
<td></td>
<td>×</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
When simple problems are hard, we should try to find good approximation heuristics. A $\varrho$-approximation is an algorithm whose output is never more than a factor $\varrho$ times the optimum solution.

Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).

**Theorem 1:**

Let $G = (V,E,w)$ be a DAG of sequential tasks, $p$ the number of processors, and $\sigma_p$ a list-schedule of $G$ on $p$ processors.

$$C_{\text{max}}(\sigma_p) \leq \left(\frac{2}{p} - \frac{1}{p}\right) C^*_{\text{max}}(p).$$

Most of the time, list-heuristics are based on the critical path.
List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A \( \varrho \)-approximation is an algorithm whose output is never more than a factor \( \varrho \) times the optimum solution.

Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).

Any strategy that does not let on purpose a processor idle is efficient [Coffman76]. Such a schedule is called list-schedule.

**Theorem 1: Coffman.**

Let \( G = (V, E, w) \) be a DAG of sequential tasks, \( p \) the number of processors, and \( \sigma_p \) a list-schedule of \( G \) on \( p \) processors.

\[
C_{\text{max}}(\sigma_p) \leq \left(2 - \frac{1}{p}\right) C_{\text{max}}^*(p).
\]
List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A $\varrho$-approximation is an algorithm whose output is never more than a factor $\varrho$ times the optimum solution. Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).

Any strategy that does not let on purpose a processor idle is efficient [Coffman76]. Such a schedule is called list-schedule.

**Theorem 1: Coffman.**

Let $G = (V, E, w)$ be a DAG of sequential tasks, $p$ the number of processors, and $\sigma_p$ a list-schedule of $G$ on $p$ processors.

$$C_{\text{max}}(\sigma_p) \leq \left(2 - \frac{1}{p}\right) C_{\text{max}}^*(p).$$

Most of the time, list-heuristics are based on the critical path.
Proof.

$$p$$

$$C_{\text{max}}(\sigma_p)$$

Therefore, $$\text{Idle} \leq (p - 1)w(\Phi)$$ for some $$\Phi$$. Hence,

$$pC_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1)w(\Phi) + \text{Seq} \leq (p - 1)C^*_{\text{max}}(p) + pC^*_{\text{max}}(p) = (2p - 1)C^*_{\text{max}}(p)$$
List Scheduling: proving the Coffman result

Proof.

Therefore, \( \text{Idle} \leq (p - 1)w(\Phi) \) for some \( \Phi \).

Hence, \( p.C_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1)w(\Phi) + \text{Seq} \leq (p - 1)C^*_{\text{max}}(p) + p.C^*_{\text{max}}(p) = (2p - 1)C^*_{\text{max}}(p) \).
Proof.

Let $\sigma_p$ be the schedule obtained from the list scheduling algorithm. We want to show that $C_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1)w(\Phi) + \text{Seq} \leq (p - 1)C^\ast_{\text{max}}(p) + pC^\ast_{\text{max}}(p) = 2pC^\ast_{\text{max}}(p)$.

Here, $\text{Idle}$ represents the idle time before the first job starts, $\text{Seq}$ is the sequence length, and $w(\Phi)$ is the processing time of the longest job in the schedule. The inequality follows from the fact that the list scheduling algorithm ensures that all jobs are processed in the order of their arrival times, and the idle time is minimized. The final expression shows that the maximum completion time is bounded by a constant multiple of the optimal makespan for $p$ processors.
List Scheduling: proving the Coffman result

Proof.

\[ p \quad C_{\text{max}}(\sigma_p) \]

Therefore,

\[ \text{Idle} \leq (p-1)w(\Phi) \]

for some \( \Phi \). Hence,

\[ p.C_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p-1)w(\Phi) + \text{Seq} \leq (p-1)C^*_{\text{max}}(p) + p.C^*_{\text{max}}(p) = (2p-1)C^*_{\text{max}}(p) \]
Proof.

\[ \text{Idle} \leq (p - 1) \cdot w(\Phi) \]

Therefore,

\[ p \cdot C_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1) \cdot w(\Phi) + \text{Seq} \leq (p - 1) C^*_{\text{max}}(p) + p \cdot C^*_{\text{max}}(p) = (2p - 1) C^*_{\text{max}}(p) \]
Proof.

\[ C_{\text{max}}(\sigma_p) \]

\[ \text{Idle} \leq (p - 1)w(\Phi) \]

Therefore, for some \( \Phi \)

\[ pC_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1)w(\Phi) + \text{Seq} \leq (p - 1)C^*_{\text{max}}(p) + pC^*_{\text{max}}(p) = (2p - 1)C^*_{\text{max}}(p) \]
Proof.

Therefore, \( \text{Idle} \leq (p - 1).w(\Phi) \) for some \( \Phi \).

Hence, \( p.C_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1).w(\Phi) + \text{Seq} \leq (p - 1)C^*_{\text{max}}(p) + p.C^*_{\text{max}}(p) = (2p - 1)C^*_{\text{max}}(p) \).
Proof.

\[ C_{\text{max}}(\sigma_p) \]

Therefore, 
\[ \text{Idle} \leq (p - 1)w(\Phi) \]

Hence, 
\[ p.C_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1)w(\Phi) + \text{Seq} \leq (p - 1)C^*_{\text{max}} + p.C^*_{\text{max}} = (2p - 1)C^*_{\text{max}} \]
List Scheduling: proving the Coffman result

Proof.

\[ \text{Therefore,} \quad \text{Idle} \leq (p - 1) \cdot \omega(\Phi) \]

Hence,

\[ p \cdot C_{\text{max}}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1) \cdot \omega(\Phi) + \text{Seq} \leq (p - 1) C_{\ast \text{max}}(p) + p \cdot C_{\ast \text{max}}(p) = (2p - 1) C_{\ast \text{max}}(p) \]
Proof.

Therefore, $Idle \leq (p - 1).w(\Phi)$ for some $\Phi$
List Scheduling: proving the Coffman result

Proof.

Therefore, \( \text{Idle} \leq (p - 1).w(\Phi) \) for some \( \Phi \)

Hence,

\[
p.C_{\max}(\sigma_p) = \text{Idle} + \text{Seq} \leq (p - 1)w(\Phi) + \text{Seq} \\
\leq (p - 1)C^*_\max(p) + p.C^*_\max(p) = (2p - 1)C^*_\max(p)
\]
List Scheduling: proving the Coffman result

One can actually prove that this bound cannot be improved.

Proof.

\[ p - 1 \left\{ \begin{array}{c}
K(p - 1) \\
K(p - 1) \\
\vdots \\
1 \\
\end{array} \right\} \rightarrow \begin{array}{c}
\vdots \\
K \\
K(p - 1) \\
\end{array} \]
List Scheduling: proving the Coffman result

One can actually prove that this bound cannot be improved.

Proof.

\[
K(p - 1) = K(p - 1)
\]

\[
2K(p - 1) + K = K(2p - 1)
\]
One can actually prove that this bound cannot be improved.

Proof.

\[
1 + K + K(p - 1) = Kp + 1
\]
One can actually prove that this bound cannot be improved.

**Proof.**

\[
p - 1 \quad K(p - 1) \quad 1 \quad p \quad K(p - 1)
\]

\[
1 + K + K(p - 1) = Kp + 1
\]

\[
\varrho \geq \frac{K(2p - 1)}{Kp + 1} \quad \xrightarrow{K \to \infty} \quad \frac{2p - 1}{p}
\]
List scheduling Anomalies

\[ MS = 19 \]
List scheduling Anomalies

Introduction to Scheduling

Batch Scheduling 51 / 80
Let us assume we have \( n \) independent rigid jobs \( J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n) \) and \( m \) machines.

Let us denote by \( T^* \) the optimal makespan for this instance.
Let us assume we have \( n \) independent rigid jobs \( J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n) \) and \( m \) machines.

Let us denote by \( T^* \) the optimal makespan for this instance.

Let us consider a list schedule of makespan \( T \). Let us denote by \( q(t) \) the number of active processors at time \( t \).

We have \( \forall t_1, t_2 \in [0, T] : t_1 \leq t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m \)

(otherwise, the tasks running at time \( t_2 \) could have been run at time \( t_1 \)).
List Scheduling for Parallel Rigid Tasks

Let us assume we have \( n \) **independent** rigid jobs \( J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n) \) and \( m \) machines.

Let us denote by \( T^* \) the optimal makespan for this instance.

Let us consider a list schedule of makespan \( T \). Let us denote by \( q(t) \) the number of active processors at time \( t \).

We have \( \forall t_1, t_2 \in [0, T] : t_1 \leq t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m \) (otherwise, the tasks running at time \( t_2 \) could have been run at time \( t_1 \)).

Let us assume that \( T > 2T^* \). Then we have:

\[
mT^* \geq \sum_i q_ip_i = \int_0^T q(t) = \int_0^{2T^*} q(t) + \int_{2T^*}^T q(t) \\
\geq \int_0^{T^*} q(t) + q(t + T^*) + \int_{2T^*}^T q(t), \text{ which is absurd.}
\]
List Scheduling for Parallel Rigid Tasks

Let us assume we have \( n \) independent rigid jobs \( J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n) \) and \( m \) machines.

Let us denote by \( T^* \) the optimal makespan for this instance.

Let us consider a list schedule of makespan \( T \). Let us denote by \( q(t) \) the number of active processors at time \( t \).

We have \( \forall t_1, t_2 \in [0, T] : t_1 \leq t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m \) (otherwise, the tasks running at time \( t_2 \) could have been run at time \( t_1 \)).

Let us assume that \( T > 2T^* \). Then we have:

\[
mT^* \geq \sum_i q_ip_i = \int_0^T q(t) = \int_0^{2T^*} q(t) + \int_{2T^*}^T q(t)
\]

**Theorem 2.**

List-scheduling has an approximation factor of 2 for minimizing the Cmax of Parallel Rigid Tasks.
Going Online

How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let \( A \) be a polynomial-time \( \varrho \)-approximation for \( \langle P | size_j | C_{\text{max}} \rangle \). Based on \( A \), we can build a \( 2 \varrho \)-competitive polynomial-time online clairvoyant algorithm for \( \langle P | size_j, r_j | C_{\text{max}} \rangle \).

**Proof.**

Let us look at the schedule produced by \( A \) on an instance \( I \).

- release of \( S_0 \) jobs

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>
How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $A$ be a polynomial-time $\varrho$-approximation for $\langle P|\text{size}_j|C_{\text{max}} \rangle$. Based on $A$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|\text{size}_j, r_j|C_{\text{max}} \rangle$.

**Proof.**

Let us look at the schedule produced by $A$ on an instance $\mathcal{I}$.
How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $\mathcal{A}$ be a polynomial-time $\varrho$-approximation for \( \langle P | \text{size}^j | C_{\text{max}} \rangle \). Based on $\mathcal{A}$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for \( \langle P | \text{size}^j, r^j | C_{\text{max}} \rangle \).

**Proof.**

Let us look at the schedule produced by $\mathcal{A}$ on an instance $I$. 

![Diagram showing schedule production](image)
How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $A$ be a polynomial-time $\varrho$-approximation for $\langle P|size_j|C_{\text{max}} \rangle$. Based on $A$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j,r_j|C_{\text{max}} \rangle$.

**Proof.**

Let us look at the schedule produced by $A$ on an instance $\mathcal{I}$.
How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $A$ be a polynomial-time $\varrho$-approximation for $\langle P|size_j|C_{max}\rangle$. Based on $A$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{max}\rangle$.

**Proof.**

Let us look at the schedule produced by $A$ on an instance $I$. 

![Diagram showing the schedule produced by $A$]

0 schedule $S_0$ $F_0$ schedule $S_1$ $F_1$ ... $F_{k-2}$ schedule $S_k-F_{k-1}$
How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $\mathcal{A}$ be a polynomial-time $\varrho$-approximation for $\langle P|size_j|C_{\text{max}} \rangle$. Based on $\mathcal{A}$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\text{max}} \rangle$.

**Proof.**

Let us look at the schedule produced by $\mathcal{A}$ on an instance $\mathcal{I}$.
Going Online

How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $A$ be a polynomial-time $\varrho$-approximation for $\langle P|size_j|C_{\text{max}} \rangle$. Based on $A$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\text{max}} \rangle$.

**Proof.**

Consider $\mathcal{I}'$ where $S_k$ jobs are released at time $F_{k-2}$. We have:

$$C^*_\text{max}(\mathcal{I}') \leq C^*_\text{max}(\mathcal{I}).$$
How can we use the previous result when going online?

**Theorem 3:** [Shmoys91].

Let $A$ be a polynomial-time $\varrho$-approximation for $\langle P|size_j|C_{\text{max}} \rangle$. Based on $A$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\text{max}} \rangle$.

**Proof.**

Consider $\mathcal{I}'$ where $S_k$ jobs are released at time $F_{k-2}$. We have:

$$C_{\text{max}}'(\mathcal{I}') \leq C_{\text{max}}(\mathcal{I}).$$

$$F_{k-2} + F_k - F_{k-1} \leq \varrho C_{\text{max}}'(\mathcal{I}')$$
How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let \( A \) be a polynomial-time \( \varrho \)-approximation for \( \langle P | \text{size}_j | C_{\text{max}} \rangle \). Based on \( A \), we can build a \( 2\varrho \)-competitive polynomial-time online clairvoyant algorithm for \( \langle P | \text{size}_j, r_j | C_{\text{max}} \rangle \).

**Proof.**

Consider \( \mathcal{I}' \) where \( S_k \) jobs are released at time \( F_{k-2} \). We have:

\[
C^*_{\text{max}}(\mathcal{I}') \leq C^*_{\text{max}}(\mathcal{I}).
\]

- \( F_{k-2} + F_k - F_{k-1} \leq \varrho C^*_\text{max}(\mathcal{I}') \)
- \( F_{k-1} - F_{k-2} \leq \varrho C^*_\text{max}(\mathcal{I}') \)
Going Online

How can we use the previous result when going online?

Theorem 3: [Shmoys91].

Let $A$ be a polynomial-time $\varrho$-approximation for $\langle P | size_j | C_{max} \rangle$. Based on $A$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P | size_j, r_j | C_{max} \rangle$.

Proof.

Consider $\mathcal{I}'$ where $S_k$ jobs are released at time $F_{k-2}$. We have:

$$C^*_\text{max}(\mathcal{I}') \leq C^*_\text{max}(\mathcal{I}).$$

- $F_{k-2} + F_k - F_{k-1} \leq \varrho C^*_\text{max}(\mathcal{I}')$
- $F_{k-1} - F_{k-2} \leq \varrho C^*_\text{max}(\mathcal{I}')$

Hence $F_k \leq 2\varrho C^*_\text{max}(\mathcal{I}') \leq 2\varrho C^*_\text{max}(\mathcal{I})$
How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let \( A \) be a polynomial-time \( \varrho \)-approximation for \( \langle P | \text{size}_j | C_{\text{max}} \rangle \). Based on \( A \), we can build a \( 2\varrho \)-competitive polynomial-time online clairvoyant algorithm for \( \langle P | \text{size}_j, r_j | C_{\text{max}} \rangle \).

- There is a PTAS for \( \langle Q || C_{\text{max}} \rangle \). Hence, there is an \( (2 + \varepsilon) \)-competitive online clairvoyant algorithm for \( \langle Q | r_j | C_{\text{max}} \rangle \).
Going Online

How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $A$ be a polynomial-time $\rho$-approximation for $\langle P|size_j|C_{\text{max}} \rangle$. Based on $A$, we can build a $2\rho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j,r_j|C_{\text{max}} \rangle$. 

- There is a PTAS for $\langle Q||C_{\text{max}} \rangle$. Hence, there is an $(2 + \varepsilon)$-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\text{max}} \rangle$.
- There is a 2 approximation $\langle Q||C_{\text{max}} \rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\text{max}} \rangle$. 

A. Legrand (CNRS-LIG) INRIA-MESCAL

Introduction to Scheduling

Batch Scheduling 53 / 80
Going Online

How can we use the previous result when going online?

Theorem 3: [Shmoys91].

Let $A$ be a polynomial-time $\varrho$-approximation for $\langle P|\text{size}_j|C_{\text{max}} \rangle$. Based on $A$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|\text{size}_j, r_j|C_{\text{max}} \rangle$.

- There is a PTAS for $\langle Q||C_{\text{max}} \rangle$. Hence, there is an $(2 + \varepsilon)$-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\text{max}} \rangle$.
- There is a 2 approximation $\langle Q||C_{\text{max}} \rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\text{max}} \rangle$.
- There is a 2 approximation $\langle P|\text{size}_j|C_{\text{max}} \rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|\text{size}_j|C_{\text{max}} \rangle$. 
Going Online

How can we use the previous result when going online?

**Theorem 3: [Shmoys91].**

Let $\mathcal{A}$ be a polynomial-time $\varrho$-approximation for $\langle P|size_j|C_{\text{max}} \rangle$. Based on $\mathcal{A}$, we can build a $2\varrho$-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\text{max}} \rangle$.

- There is a PTAS for $\langle Q||C_{\text{max}} \rangle$. Hence, there is an $(2 + \varepsilon)$-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\text{max}} \rangle$.
- There is a 2 approximation $\langle Q||C_{\text{max}} \rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\text{max}} \rangle$.
- There is a 2 approximation $\langle P|size_j|C_{\text{max}} \rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|size_j|C_{\text{max}} \rangle$.
- Actually, by doing a slightly finer analysis, one can show that the list-scheduling algorithm is a $(2 - 1/m)$-competitive non-clairvoyant algorithm for $\langle P|r_j|C_{\text{max}} \rangle$. 
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Jobs arrive one after the other and are scheduled at arrival.
Jobs arrive one after the other and are scheduled at arrival.
Jobs arrive one after the other and are scheduled at arrival.
Jobs arrive one after the other and are scheduled at arrival.
FCFS = simplest scheduling option

Fragmentation $\leadsto$ need for backfilling
First Come First Served

- FCFS = simplest scheduling option
- Fragmentation $\sim$ need for backfilling
First Come First Served

- FCFS = simplest scheduling option
- Fragmentation $\sim$ need for backfilling
Backfilling: Question

- Which job(s) should be picked for promotion through the queue?
- Many heuristics are possible
- Two have been studied in detail
  - EASY
  - Conservative Back Filling (CBF)
- In practice EASY (or variants of it) is used, while CBF is not.
- Although, OAR, a recently proposed batch scheduler implements CBF.
Outline

1 Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2 Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3 Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4 Gang Scheduling as an Alternative
   - Principles
EASY Backfilling

Extensible Argonne Scheduling System
Maintain only one reservation, for the first job in the queue.
Definitions:

Shadow time time at which the first job in the queue starts execution

Extra nodes number of nodes idle when the first job in the queue starts execution

1. Go through the queue in order starting with the 2nd job.
2. Backfill a job if it will terminate by the shadow time, or it needs less than the extra nodes.
Property:

- The first job in the queue will never be delayed by backfilled jobs
Property:
- The first job in the queue will never be delayed by backfilled jobs
Property:

- The first job in the queue will never be delayed by backfilled jobs.
Property:

- The first job in the queue will never be delayed by backfilled jobs
Property:

- The first job in the queue will never be delayed by backfilled jobs
Property:

- The first job in the queue will never be delayed by backfilled jobs.
- BUT, other jobs may be delayed infinitely!
Property:

- The first job in the queue will never be delayed by backfilled jobs.
- BUT, other jobs may be delayed infinitely!
Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!
Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!
EASY Properties

Unbounded Delay. ▶ The first job in the queue will never be delayed by backfilled jobs
▷ BUT, other jobs may be delayed infinitely!

No Starvation. ▶ Delay of first job is bounded by runtime of current jobs
▷ When the first job finishes, the second job becomes the first job in the queue
▷ Once it is the first job, it cannot be delayed further

Other approach. ▶ Conservative Backfilling. EVERY job has a reservation. A job may be backfilled only if it does not delay any other job ahead of it in the queue.
▷ Fixes the unbounded delay problem that EASY has. More complicated to implement (The algorithm must find holes in the schedule) though.
▷ EASY favors small long jobs and harms large short jobs.
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
When Does Backfilling Happen?

Possibly when

- A new job arrives

Trade-off:

- Provide a conservative estimate: your job goes through the queue faster (may be backfilled)
- Provide a loose estimate: your job will not be killed

Are estimates accurate?
When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early
When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).
When Does Backfilling Happen?

Possibly when

► A new job arrives
► The first job in the queue starts
► When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).

Trade-off:

► provide a conservative estimate: you go through the queue faster (may be backfilled)
► provide a loose estimate: your job will not be killed
When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).

Trade-off:

- provide a conservative estimate: you go through the queue faster (may be backfilled)
- provide a loose estimate: your job will not be killed

Are estimates accurate?
All of this is great, but how do we know what a “good” schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs:

1. **Turn-around time or flow (Wait time + Run time).**
   - Job 1 needs 1h of compute time and waits 1s
   - Job 2 needs 1s of compute time and waits 1h
   - Clearly Job 1 is really happy, and Job 2 is not happy at all

2. **Wait time (equivalent to “user happiness”).**
   - Job 1 asks for 1 nodes and waits 1 h
   - Job 2 asks for 512 nodes and waits 1h
   - Again, Job 1 is unhappy while Job 2 is probably sort of happy.

3. **Slowdown or Stretch (turn-around time divided by turn-around time if alone in the system).**
   - Doesn’t really take care of the small/large problem. Could think of some scaling, but unclear!
All of this is great, but how do we know what a “good” schedule is? FCFS, EASY, CFB, Random?
What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs.

1. Turn-around time or flow (Wait time + Run time).
   - Job 1 needs 1h of compute time and waits 1s
   - Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all.
How Good is the Schedule?

All of this is great, but how do we know what a “good” schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs.

1. **Turn-around time or flow** (Wait time + Run time).
   - Job 1 needs 1h of compute time and waits 1s
   - Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all

2. **Wait time** (equivalent to “user happiness”)
   - Job 1 asks for 1 nodes and waits 1 h
   - Job 2 asks for 512 nodes and waits 1h
   Again, Job 1 is unhappy while Job 2 is probably sort of happy.

We need a metric that represents happiness for small, large, short, long jobs.
How Good is the Schedule?

All of this is great, but how do we know what a “good” schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

1. Turn-around time or flow (Wait time + Run time).
   - Job 1 needs 1h of compute time and waits 1s
   - Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all

2. Wait time (equivalent to “user happiness”)
   - Job 1 asks for 1 nodes and waits 1 h
   - Job 2 asks for 512 nodes and waits 1h
   Again, Job 1 is unhappy while Job 2 is probably sort of happy.
   We need a metric that represents happiness for small, large, short, long jobs.

3. Slowdown or Stretch (turn-around time divided by turn-around time if alone in the system)
   Doesn’t really take care of the small/large problem. Could think of some scaling, but unclear!
Now we have a few metrics we can consider
We can run simulations of the scheduling algorithms, and see how they fare.
We need to test these algorithms in representative scenarios Supercomputer/cluster traces. Collect the following for long periods of time:

- Time of submission
- How many nodes asked
- How much time asked
- How much time was actually used
- How much time spent in the queue

Uses of the traces:
1. Drive simulations
2. Come up with models of user behaviors
A type of experiments that people have done: replace user estimate by $f$ times the actual run time
Possible to improve performance by multiplying user estimates by 2!

<table>
<thead>
<tr>
<th></th>
<th>EASY</th>
<th>CBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Slowdown</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KTH</td>
<td>-4.8%</td>
<td>-23.0%</td>
</tr>
<tr>
<td>CTC</td>
<td>-7.9%</td>
<td>-18.0%</td>
</tr>
<tr>
<td>SDSC</td>
<td>+4.6%</td>
<td>-14.2%</td>
</tr>
<tr>
<td>Mean Response time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KTH</td>
<td>-3.3%</td>
<td>-7.0%</td>
</tr>
<tr>
<td>CTC</td>
<td>-0.9%</td>
<td>-1.6%</td>
</tr>
<tr>
<td>SDSC</td>
<td>-1.6%</td>
<td>-10.9%</td>
</tr>
</tbody>
</table>
These are all heuristics.

They are not specifically designed to optimize the metrics we have designed.

It is difficult to truly understand the reasons for the results.

But one can derive some empirical wisdom.

One of the reasons why one is stuck with possibly obscure heuristics is that we’re dealing with an on-line problem: We don’t know what happens next.

We cannot wait for all jobs to be submitted to make a decision. But we can wait for a while, accumulate jobs, and schedule them together.
Batch Schedulers are what we’re stuck with at the moment. They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn’t finish within 1 hour and gets killed!!
Batch Schedulers are what we’re stuck with at the moment. They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn’t finish within 1 hour and gets killed!!

A lot of research, a few things happening “in the field”. When you go to a company that has clusters (like most of them), they typically have a job scheduler, so it’s good to have some idea of what it is.
Batch Schedulers are what we’re stuck with at the moment. They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn’t finish within 1 hour and gets killed!!

A lot of research, a few things happening “in the field”. When you go to a company that has clusters (like most of them), they typically have a job scheduler, so it’s good to have some idea of what it is.

A completely different approach is **gang scheduling**, which we discuss next.
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles

A. Legrand (CNRS-LIG) INRIA-MESCAL

Introduction to Scheduling

Gang Scheduling 69 / 80
Introduction to Scheduling

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Gang Scheduling: Basis

- All processes belonging to a job run at the same time (the term **gang** denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid **coordinated** context switching.
- It is possible to **suspend/preempt** jobs arbitrarily.
All processes belonging to a job run at the same time (the term *gang* denotes all processors within a job).

Each process runs alone on each processor.

BUT: there is rapid *coordinated* context switching.

It is possible to *suspend/preempt* jobs arbitrarily ~ May allow more flexibility to optimize some metrics.
Gang Scheduling: Basis

- All processes belonging to a job run at the same time (the term *gang* denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid *coordinated* context switching.
- It is possible to *suspend/preempt* jobs arbitrarily \( \sim \) May allow more flexibility to optimize some metrics.
- If processing times are not known in advance (or grossly erroneous), preemption can help short jobs that would be “stuck” behind a long job.
- Should improve machine utilization.
Gang Scheduling: an Example

- A 128 node cluster.
- A running 64-node job.
- A 32-node job and a 128-node job are queued.

Should the 32-node job be started?

Space-Sharing
- short
- 32-node job
- long
- 32-node job

Time-Sharing

More uniform slowdown, better resource usage.

A. Legrand (CNRS-LIG) INRIA-MESCAL
Introduction to Scheduling
Gang Scheduling
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency (Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
- Typically not used in production HPC systems (batch scheduling is preferred).
- Some implementations (MOSIX, Kerighed).
Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency (Frequent cache flushing).

RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).

Typically not used in production HPC systems (batch scheduling is preferred).

Some implementations (MOSIX, Kerighed).
Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency (Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency (frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
- Typically not used in production HPC systems (batch scheduling is preferred).
Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency (Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
- Typically not used in production HPC systems (batch scheduling is preferred).
- Some implementations (MOSIX, Kerighed).
1 Modeling Applications, General Notions
   • Introducing Fundamental Notions Through the Matrix Product Example
   • Adaptive Parallel Programs
   • Task Graphs and Parallel Tasks From Outer Space

2 Defining a Scheduling Problem
   • Rules of the Game
   • Criteria: How Do You Win the Game?
   • Analysis Method
   • Graham Notation

3 Batch Scheduling
   • Principles
   • Theoretical results
   • Basic idea: FCFS + Backfilling
   • EASY
   • How Good is the Schedule?

4 Gang Scheduling as an Alternative
   • Principles
So it seems we’re stuck with batch scheduling. Why don’t we like Batch Scheduling?

Because queue waiting times
depend on the status of the queue
depend on the scheduling algorithm used
depends on all sorts of configuration parameters set by system administrator
depends on future job completions!

etc.

So I submit my job and then it’s in limbo somewhere, which is eminently annoying to most users.

That is why there is more and more demand for reservation support.
Users build (badly?) the schedule by themselves.
So it seems we’re stuck with batch scheduling. Why don’t we like Batch Scheduling? Because queue waiting times are difficult to predict.

- depends on the status of the queue
- depends on the scheduling algorithm used
- depends on all sorts of configuration parameters set by system administrator
- depends on future job completions!
- etc.

So I submit my job and then it’s in limbo somewhere, which is eminently annoying to most users.
So it seems we’re stuck with batch scheduling. Why don’t we like Batch Scheduling? Because queue waiting times are difficult to predict.

- depends on the status of the queue
- depends on the scheduling algorithm used
- depends on all sorts of configuration parameters set by system administrator
- depends on future job completions!
- etc.

So I submit my job and then it’s in limbo somewhere, which is eminently annoying to most users. That is why there is more and more demand for reservation support. Users build (badly?) the schedule by themselves.
Outline

1. Modeling Applications, General Notions
   - Introducing Fundamental Notions Through the Matrix Product Example
   - Adaptive Parallel Programs
   - Task Graphs and Parallel Tasks From Outer Space

2. Defining a Scheduling Problem
   - Rules of the Game
   - Criteria: How Do You Win the Game?
   - Analysis Method
   - Graham Notation

3. Batch Scheduling
   - Principles
   - Theoretical results
   - Basic idea: FCFS + Backfilling
   - EASY
   - How Good is the Schedule?

4. Gang Scheduling as an Alternative
   - Principles
Grids result from the collaboration of many Universities/Computing Centers.
Everyone runs its own Batch Scheduler that cannot be bypassed.
How to decide where we should submit our jobs?

When in doubt, a brute-force approach is to:
- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

What happens if everybody does this?

Other issues:
- File Staging?
- Load Balancing between sites?
Grids result from the collaboration of many Universities/Computing Centers. Everyone runs its own Batch Scheduler that cannot be bypassed. How to decide where we should submit our jobs? When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.
Grids result from the collaboration of many Universities/Computing Centers. Everyone runs its own Batch Scheduler that cannot be bypassed. How to decide where we should submit our jobs? When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

What happens if everybody does this?
Grids result from the collaboration of many Universities/Computing Centers. Everyone runs its own Batch Scheduler that cannot be bypassed. How to decide where we should submit our jobs? When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

What happens if everybody does this?

Other issues:

- File Staging?
- Load Balancing between sites?
A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let’s try a few natural scheduling strategies. We denote by $a_i$ the time at which $P_i$ is available (at the beginning $a_i = 0$ for all $P_i$):

- **Min-Min**: Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each $J_j$ and choose the one with the smallest $C_j$. Update the corresponding $a_i$ (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

- **Max-Min**: Choose $J_j$ with the largest $C_j$ and update the corresponding $a_i$ (its best host) accordingly.

- **Sufferage $S_j$**: is the difference between the best completion time of $J_j$ and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

Problem: How do you get an estimate of $p_{i,j}$?
A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let's try a few natural scheduling strategies. We denote by $a_i$ the time at which $P_i$ is available (at the beginning $a_i = 0$ for all $P_i$):

**Min-Min** Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each $J_j$ and choose the one with the smallest $C_j$. Update the corresponding $a_i$ (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

**Max-Min** Choose $J_j$ with the largest $C_j$ and update the corresponding $a_i$ (its best host) accordingly.

**Sufferage** $S_j$ is the difference between the best completion time of $J_j$ and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

**Problem:** How do you get an estimate of $p_{i,j}$?
A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let’s try a few natural scheduling strategies. We denote by $a_i$ the time at which $P_i$ is available (at the beginning $a_i = 0$ for all $P_i$):

**Min-Min** Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each $J_j$ and choose the one with the smallest $C_j$. Update the corresponding $a_i$ (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

**Max-Min** Choose $J_j$ with the largest $C_j$ and update the corresponding $a_i$ (its best host) accordingly.
A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let’s try a few natural scheduling strategies. We denote by $a_i$ the time at which $P_i$ is available (at the beginning $a_i = 0$ for all $P_i$):

**Min-Min** Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each $J_j$ and choose the one with the smallest $C_j$. Update the corresponding $a_i$ (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

**Max-Min** Choose $J_j$ with the largest $C_j$ and update the corresponding $a_i$ (its best host) accordingly.

**Sufferage** $S_j$ is the difference between the best completion time of $J_j$ and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

Problem: How do you get an estimate of $p_{i,j}$?
Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).

Let’s try a few natural scheduling strategies. We denote by $a_i$ the time at which $P_i$ is available (at the beginning $a_i = 0$ for all $P_i$):

**Min-Min** Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each $J_j$ and choose the one with the smallest $C_j$. Update the corresponding $a_i$ (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

**Max-Min** Choose $J_j$ with the largest $C_j$ and update the corresponding $a_i$ (its best host) accordingly.

**Sufferage** $S_j$ is the difference between the best completion time of $J_j$ and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

**Problem:** How do you get an estimate of $p_{i,j}$?
Batch schedulers are complex pieces of software that are used in practice.

A lot of experience on how they work and how to use them.

But ultimately everybody knows they are an imperfect solution.

Many view the lack of theoretical foundations as a big problem.

Some just don’t care…

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

Bibliography


