Parallel identical processor scheduling with weighted completion time

Simone Cavana
Matricola 138519

Febbraio 2021

Indice

1 Introduzione

In questa prima parte sarà effettuato uno studio del problema dello scheduling di job su macchine parallele identiche attraverso alcuni articoli dopodiché verrà proposta l'idea di sviluppo basata su un articolo di riferimento e la conseguente implementazione tramite codice.

Il problema dello scheduling di task su macchine parallele è un argomento ben dettagliato e studiato in letteratura e viene descritto come $P\|\sum w_j C_j$. In generale si hanno dei task di lavoro da assegnare ad una serie di macchine parallele identiche e si vuole minimizzare la somma dei tempi di completamento dei task, ciascuno pesato. Inoltre si ha che per ogni macchina può eseguire solamente un job per istante e non è ammessa preemption, ovvero quando un job inizia deve arrivare alla fine. In particolare:

$$
\begin{aligned}
M &= \text{set di m macchine identiche} \\
J &= \text{set di n job} \\
p_j &= \text{tempo d'esecuzione di un job} \\
w_j &= \text{peso di un job} \\
C_j &= \text{tempo di completamento di un job in una tabella di marcia dei tempi}
\end{aligned}
$$

L'output desiderato è una tabella di "marcia" che ci indichi quali task eseguano su quali macchine in un determinato range di tempo. Questo problema risulta essere $\mathcal{NP}$-hard nel caso in cui il numero di macchine m sia variabile e nel corso di Applicazione della Ricerca Operativa viene presentato il problema e risolto tramite l'utilizzo di Column Generation e B&B come descritto negli articoli [1], [2] e [3].

2 Articolo di riferimento

La soluzione implementata utilizza un metodo esatto basato su un algoritmo di Column Generation fortemente ispirato a quello di Van Den Akker [3] per risolvere il rilassamento continuo del problema e trovare un lower bound valido. Questo algoritmo viene inizializzato tramite l'euristica "The Randomized List Scheduling Heuristic" e infine verrà dettagliato un paragone con i risultati ottenuti nell'articolo di riferimento su differenti istanze di benchmark per comprendere le caratteristiche in termini di qualità della soluzione e tempistiche.

La formulazione del modello viene descritta come un problema di set-covering con un numero esponenziale di variabili binarie. **Definiamo una schedula s come un insieme di job ammissibili assegnabile ad una qualsiasi macchina m**. A questo punto bisogna tenere conto delle seguenti importanti considerazioni:

(i) i job devono essere eseguiti in modo contiguo su ogni macchina dall'istante zero in avanti, senza la presenza di idle finchè ci sono dei job da schedulare.

(ii) è utile considerare una finestra di tempo in cui è possibile schedulare l'ultimo job su ogni macchina composta da $[H_{min}, H_{max}]$, dove $H_{min} = (\sum_{j\in J} p_j - (m-1)p_{max})/m$, $H_{max} = (\sum_{j\in J} p_j + (m-1)p_{max})/m$ e $p_{max} = \max_{j\in J} p_j$.

(iii) viene fatto uso dell'**ordinamento di Smith** che ci indica come ordinare i job sulle singole macchine. Ovvero viene effettuato un ordinamento in ordine decrescente sulla base dell'importanza di ogni job rispetto al suo tempo di esecuzione w_j/p_j. Questo comporta la trasformazione del problema iniziale in un problema di partizionamento.

(iv) se $w_j \geq w_k$ e $p_j \leq p_k$ allora esiste una schedula ottima dove il job J_j inizia ad eseguire prima del job J_k.

Bisogna inoltre tenere in considerazione dell'esistenza di un algoritmo di programmazione dinamica che permette la risoluzione in $\mathcal{O}(n(\sum_{j=1}^{n} p_j)^{m-1})$ sia per il tempo che per l'archiviazione; questo non risulta sufficiente soprattutto quando il numero di macchine m cresce.

Presentiamo ora il modello principale, definito come **set-covering**, che sfrutta l'utilizzo delle migliori schedule da assegnare alle m macchine identiche:

$$s = \text{schedula ammissibile, } s \in S$$
$$S = \text{set delle schedule disponibili}$$
$$a_{js} = \begin{cases} 1 & \text{se il job j è assegnato alla schedula s,} \\ 0 & \text{altrimenti} \end{cases}$$
$$C_j(s) = \text{tempo di completamento di un job in una schedula s} = \sum_{k=1}^{j} a_{ks} p_k$$
$$c_s = \text{costo di una schedula} = \sum_{j \in J} w_j a_{js} C_j(s) = \sum_{j \in J} w_j a_{js} \left[\sum_{k=1}^{j} a_{ks} p_k \right]$$
$$x_s = \begin{cases} 1 & \text{se la schedula s è selezionata,} \\ 0 & \text{altrimenti} \end{cases}$$

$$\begin{aligned} \min \quad & \sum_{s \in S} c_s x_s \\ \text{s.t.} \quad & \sum_{s \in S} x_s = m & (1) \\ & \sum_{s \in S} a_{js} x_s = 1, \quad j \in J & (2) \\ & x_s \in \{0,1\}, \qquad s \in S & (3) \end{aligned}$$

Dove il vincolo (1) insieme alle condizioni di integralità (3) assicurano che siano selezionate esattamente m schedule, ovvero tante quante il numero di macchine. Il vincolo (2) invece descrive che ogni lavoro venga eseguito esattamente una volta. Si noti che il segno di uguaglianza nelle condizioni (1) e (2) può essere cambiato in "minore di" e "maggiore di", rispettivamente, senza perdere la validità della formulazione. Dato l'elevato numero di colonne $|S|$ che vengono a generarsi con questa formulazione ne attraverso set-covering ne attraverso il suo rilassamento continuo si può risolvere il problema sfruttando tutte le colonne ammissibili a priori. Per questo motivo introduciamo il column generation.

2.1 Column Generation

Il column generation è un metodo che risolve il rilassamento continuo dei modelli estesi. Nel nostro caso d'uso andremo appunto a rilassare il problema del set covering e, dato un sottoinsieme $\bar{S} \subseteq S$ delle colonne di partenza scelte dall'euristica descritta in 2.3, andremo a risolvere il rilassamento continuo tramite column generation (non dobbiamo forzare l'upper bound di x_s a 1 perchè lo troviamo già incluso nel vincolo (2)). Per quanto riguarda invece la generazione di nuove colonne ammissibili si fa uso dell'algoritmo di Pricing che si riferisci al sottoproblema di Pricing 2.1.1. Di seguito si può trovare il *Master problem*:

$$\begin{aligned} \min \quad & \sum_{s \in S} c_s x_s \\ \text{s.t.} \quad & \sum_{s \in S} x_s = m, & (\lambda_0) \\ & \sum_{s \in S} a_{js} x_s = 1, \quad j \in J & (\lambda_j) \\ & x_s \geq 0, \qquad s \in S \end{aligned}$$

2.1.1 Pricing algorithm

Questo algoritmo risolve un problema di ottimizzazione per generare nuove colonne per il problema column generation in modo da andare a trovare delle soluzioni migliori includendo nuove colonne nello spazio di lavoro. Quello che sfrutta questo algoritmo è appunto l'utilizzo dei costi ridotti $\bar{c_s}$ che nei problemi tipici di ottimizzazione devono essere non negativi per ogni variabile per far sì di trovarsi nella soluzione ottima. Per il nostro problema i costi ridotti per ogni schedula sono così descritti (si ricavano dalla costruzione del problema duale):

$$\begin{aligned}\bar{c_s} &= c_s - \lambda_0 - \sum_{j\in J} a_{js}\lambda_j \\ &= \sum_{j\in J} w_j a_{js} \sum_{k=1}^{j} a_{ks}p_k - \lambda_0 - \sum_{j\in J} a_{js}\lambda_j \\ &= \sum_{j\in J}\left[w_j\left(\sum_{k=1}^{j} a_{ks}p_k\right) - \lambda_j\right]a_{js} - \lambda_0\end{aligned}$$

Dove λ_0 risulta essere una costante uguale su tutte le schedule e può essere trascurata. Infine vediamo che questo algoritmo verifica se una soluzione ammissibile al rilassamento continuo è ottima; se l'esito è negativo, ritorna un insieme di schedule s con $\bar{c_s} < 0$ tra cui la schedula con costo ridotto minimo. In questo particolare caso l'algoritmo di pricing viene strutturato come un algoritmo di programmazione dinamica in cui nella fase di ricorsione si ha uno stage per ogni job ed uno stato per ogni tempo di completamento. Definiamo $P(j)$ come la somma dei tempi d'esecuzione dei job fino a j mentre utilizziamo $F_j(t)$ per definire il costo ridotto minimo ricavato su tutte le schedule composte dai job fino a j e che concludono entro l'istante t (non è detto che j sia l'ultimo).

$$\begin{aligned}P(j) &= \sum_{k=1}^{j} p_k \\ F_j(t) &= \begin{cases}-\lambda_0 & \text{se } j = 0, \text{ e } t = 0, \\ \infty & \text{altrimenti}\end{cases} \qquad \textbf{\textit{INIT}}\end{aligned}$$

La ricorsione per la programmazione dinamica viene posta secondo la proprietà per cui su ogni macchina i job sono ordinati in modo crescente di indice. La schedula che realizza $F_j(t)$ può avere due possibilità: lasciare il job conclusivo J_j fuori dalla schedula oppure includerlo. Da cui per i successivi step di ricorsione $j = 1, \ldots, n,\ t = 0, \ldots, \min\{P(j), H_{max}\}$ si ha

$$F_j(t) = \begin{cases}\min\{F_{j-1}(t), F_{j-1}(t-p_j) + w_j t - \lambda_j\} & \text{se } r_j + p_j \le t \le d_j \\ F_{j-1}(t) & \text{altrimenti}\end{cases} \qquad \textbf{\textit{PASSO}}$$

$$F^* = \min_{H_{min}\le t\le H_{max}} F_n(t)$$

mentre F^* rappresenta l'ottimo che se risulta essere positivo allora abbiamo trovato la soluzione ottima al problema di ottimizzazione lineare, altrimenti bisogna inserire in base delle nuove colonne. Le colonne candidate ad essere scelte sono quelle con t per cui soddisfano $F_n(t) < 0$ e possono essere ricavate tramite **backtracing**. Per quanto riguarda il numero di colonne da aggiungere ad ogni iterazione si ha che secondo una scelta empirica le tre con t alle quali corrisponde $F_n(t)$ più negativo risulta una buona scelta.

La complessità computazionale dell'algoritmo di pricing è data dal numero di stage n in cui si hanno al massimo H_{max} stati in cui fare il semplice controllo $F_j(t)$, $\mathcal{O}(1)$. Tutto questo porta ad una complessita $\mathcal{O}(nH_{max})$ da cui si evince il fatto che se m=2 conviene sfruttare direttamente la programmazione dinamica citata all'inizio e non il column generation.

Teorema 1

If $C_j(s) = C_j \ \forall \ j \in J$ and $\forall \ s$ with $x_s^* > 0$, then the schedule obtained by processing J_j in the time interval $[C_j - p_j, C_j](j = 1, \ldots, n)$ is feasible and has minimum cost.

Ovvero se si hanno schedule in cui per ogni job in tutte le schedule ho lo stesso tempo di completamento e la soluzione al problema rilassato è ammissibile allora si riesce a ricavare la soluzione ottima intera semplicemente facendo degli scambi fra i job nelle schedule.

A questo punto, ricavata la soluzione ottima x^* al problema rilassato come descritto precedentemente, bisogna riportare la soluzione ad un valore intero valido per il problema originale di set-covering. A questo proposito abbiamo tre possibilità:

1. x^* è già intera;
2. x^* non è intera ma è rispettato il **Teorema 1**, basta quindi effettuare degli swap fra job in schedule diverse;
3. bisogna ricorrere all'utilizzo di B&B in quanto non sono rispettate le condizioni precedenti.

2.2 Branch & Bound

Il Branch & Bound è un algoritmo che mira, dato un bound minimo, ad iterare su sotto-porzioni dello spazio di partenza per controllare se in queste sotto-porzioni si migliora il lower bound dato. Nel caso in cui la soluzione trovata migliori quella di partenza dunque si procede con un nuovo branching in questa sotto-porzione, altrimenti si passa ad una nuova area di lavoro. In questo particolare caso di lavoro il bound è appunto dato da x^* mentre il branching può essere effetuato in due modi differenti: sugli intervalli di esecuzione oppure sull'immediato successore. Inoltre è noto che risulta efficace applicare il B&B a seguito del column generation solamente se le colonne generate dal pricing rispettano i vincoli imposti dalla strategia di branching.

2.2.1 Branching sugli intervalli di esecuzione

Nel caso in cui la soluzione ottima al problema master ristretto e rilassato non soddisfi il Teorema 1 si avrà almeno un job j e una schedula s con $x_s^* > 0$ che

$$\sum_{s \in S^*} C_j(s) x_s^* > \min\{C_j(s), s \in S^*\} \equiv C_j^{min}$$

tenendo conto che C_j^{min} è intero in quanto rappresenta il tempo di completamento in una schedula ammissibile. A questo punto per ogni nodo dell'albero dell'B&B si identifica il nodo frazionario con indice minore il quale viene splittato in due nodi figli:

$$C_j \leq C_j^{min} \qquad\qquad C_j \geq C_j^{min} + 1$$

Questo nella pratica viene implementando risettando la finestra di scheduling $[r_j, d_j]$ come

$$d_j \leftarrow C_j^{min} \qquad\qquad r_j \leftarrow C_j^{min} + 1 - p_j$$

ovvero la nuova deadline d_j risulta essere ristretta rispetto a quella corrente, mentre release time r_j risulta essere più grande del range minimo corrente di release. Nell'articolo viene mostrato come questo ovviamente influenza anche i predecessori $\mathcal{P}_j$ e i successori $\mathcal{S}_j$ del job in questione in quanto si può dimostrare che esiste una schedula ottima in cui i predecessori di j vengono schedulati prima di j e viceversa i successori vengono schedulati dopo. Per questo motivo viene illustrato come modificare anche le finestre temporali degli altri job.

$$\mathcal{P}_j = \{J_k | k < j, w_k \geq w_j, p_k \leq p_j\}$$
$$\mathcal{S}_j = \{J_k | k > j, w_k \leq w_j, p_k \geq p_j\}$$

$$d_k \leftarrow \min\{d_k, d_j - p_j + p_k\} \quad \forall J_k \in \mathcal{P}_j \qquad r_k \leftarrow \max\{r_k, r_j\} \quad \forall J_k \in \mathcal{S}_j$$

Nel caso in cui andando ad aggiornare le finestre temporali ci porti in istanze del problema non ammissibili allora questo branch viene tagliato e si passa al successivo. Ovviamente controllare che tutti i job siano in finestre temporali ammisibili ci porta in un problema $\mathcal{NP}$-complete e perciò sfruttiamo la condizione necessaria per cui sostituendo $\bar{d}_j \leftarrow d_j$ e andando a minimizzare $L_{max} = \max_{1 \leq j \leq n} C_j - d_j$ bisogna avere $L_{max} \leq 0$ per farsì che l'istanza sia ritenuta feasible. Questo problema di ottimizzazione è a sua volta $\mathcal{NP}$-hard ma si è scelto di lavorare sul lower bound propost da Vandevelde et al. come descritto nell'articolo di riferimento.

Può anche essere che l'attuale insieme di colonne $\bar{S}$, che si è formato risolvendo il rilassamento della programmazione lineare nel nodo precedente, non costituisca una soluzione ammissibile a causa della nuova finestra temporale. Rimuoviamo per prima cosa le colonne non ammissibili che non fanno parte della soluzione attuale al rilassamento della programmazione lineare; queste colonne possono essere cancellate impunemente. Per quanto riguarda le colonne non ammissibili che attualmente fanno parte della soluzione di programmazione lineare, aumentiamo i loro costi a un valore elevato M, tale che il valore della soluzione corrente aumenta fino a un valore maggiore dell'upper bound. Usando questo trucco, possiamo procedere con il column generation per risolvere l'istanza corrente del problema perché ci siamo assicurati che esista almeno una soluzione ammissibile. Se torniamo indietro nell'albero del B&B, riduciamo i costi di queste colonne ai loro valori reali.

2.2.2 Branching sull'immediato successore

Questa seconda regola di branching deriva da Chen e Powell [2] e sfrutta la condizione che per ogni job su ogni macchina bisogna essere a conoscenza degli immediati vicini, ovvero il job che lo precede e quello che segue. Vengono aggiunte a questo proposito due variabili fittizie, J_0 e J_{n+1}, per identificare i job all'inizio e alla fine di una schedula. A questo punto date $S_{ij} \subseteq S$ le colonne in cui i viene schedulato prima di j si può calcolare

$$y_{ij} = \sum_{s \in S_{ij}} x_s^*$$

che se risulta essere intero esiste un teorema che dice che allora è la soluzione ottima per il nodo considerato mentre se è reale andremo ad effettuare branching sul nodo più vicino a 0.5. Da cui

$$\underbrace{y_{ij} = 0}_{\text{escludo schedula con } i<j} \qquad \underbrace{y_{ij} = 1}_{i<j \text{ sempre}}$$

che ci conduce a rimuovere in entrambi i nodi le colonne che non soddisfano i vincoli (**left**: i **è** predecessore di j; **right**: i **non è** predecessore di j). Questo però ci porta a dover modificare l'algoritmo dinamico di pricing per salvarci il job che termina all'istante t; aggiungiamo quindi $B(j)(j = 1, \ldots, n, n+1)$ che risulta essere il vettore degli indici degli immediati predecessori ammissibili per il job j. Da cui si ricava

$$\bar{F}_j(t) = \begin{cases} \min\{\bar{F}_{j-1}(t), \min_{i \in B(j)}\{\bar{F}_i(t - p_j) + w_{jt} - \lambda_j\}\} & \text{se } r_j + p_j \leq t \leq d_j \\ \\ \bar{F}_{j-1}(t) & \text{altrimenti} \end{cases}$$

$$F^* = \min_{H_{min} \leq t \leq H_{max}, j \in B(n+1)} F_j(t)$$

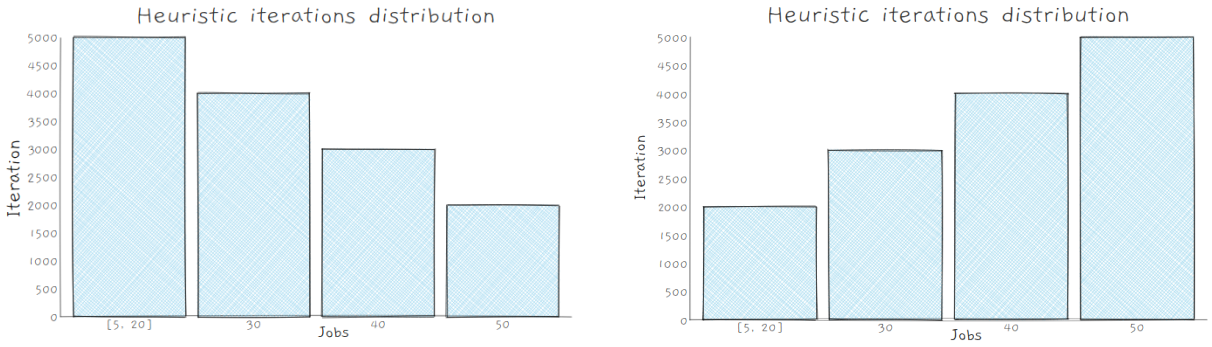
che comporta un ulteriore $\mathcal{O}(n)$ sul costo della ricorsione precedente.

2.3 The Randomized List Scheduling Heuristic

Questa euristica viene sfruttata per inizializzare il set $\bar{S}$ da utilizzare per il primo round del column generation. L'idea dell'algoritmo è di generare m schedule ad ogni iterazione assegnando n-1 lavori ordinati secondo Smith in modo casuale alle macchine, dove una macchina disponibile precedentemente ha una maggiore probabilità di ottenere il lavoro successivo in lista. In particolare, la prima macchina disponibile secondo gli istanti di tempo ha una probabilità dell'80% di ottenere il lavoro successivo, la seconda macchina disponibile ha una probabilità del 15% e la terza macchina disponibile ha una probabilità del 5%. L'ultimo lavoro viene sempre assegnato alla prima macchina disponibile. *In pratica, dopo aver ordinato n-1 job secondo l'ordinamento di Smith, ciascuno di questi viene attribuito alla prima macchina disponibile che permette di schedulare il job (con l'aggiunta di queste tre probabilità e l'assegnamento del job n-esimo alla prima macchina disponibile).*

Questa semplice e veloce euristica rispetta la considerazione (iii) ma non il rispetto delle finestre temporali $[r_j, d_j]$ che invece verrà fatto rispettare con l'algoritmo di pricing. Quindi la scelta è quella di eseguire l'euristica N volte, dove $2,000 \leq N \leq 5,000$, in base alla dimensione del benchmark. A questo punto avremo un numero di colonne pari a Nm e ne andremo a selezionare le $10m$ migliori schedule di tutte le iterazioni e su ognuna di queste verrà applicato un **neighborhood search** per migliorarle quanto possibile in modo da darle in pasto al column generation con meno lavoro da svolgere.

Dato che dall'articolo analizzato non si evince come distribuire il numero di iterazioni in funzione della dimensione dell'input vengono di seguito proposte le due opzioni che sono state analizzate. La figura a sinistra 1a rappresenta un opzione rivolta a mantenere il tempo d'esecuzione di questa parte del processo omogenea e quindi indipendente dall'istanza di input, mentre la figura 1b declina una soluzione che mira a portare una maggiore qualità nella costruzione delle colonne iniziali $\bar{S}$; questo perchè dato un numero di job più ampio andrò a realizzare un numero di colonne maggiore da cui selezionare le $10m$ migliori. Nel nostro caso è stato scelto di optare per una maggiore qualità delle colonne iniziali proposte a discapito del tempo d'esecuzione.



(a) Distribution aimed at time

(b) Distribution aimed at quality

Figura 1: Possible Heuristic Iteration Distribution

Il neighborhood di una schedula ammissibile consiste di tutte le schedule in cui i job sono ordinati secondo l'ordinamento di Smith e possono essere ottenute secondo l'insieme di operazioni (vedi figura 2):

- insert: spostare un job da una macchina ad un altra;
- swap: scambiare due job schedulati su macchine differenti.

Se si trova una schedula migliore viene selezionata come corrente e la ricerca termina quando non possono essere più trovate migliorie. Si può inoltre ridurre il carico di iterazioni facendo iterare l'euristica fino al raggiungimento di un numero totale di iterazioni o fino ad un dato valore di passi consecutivi non migliorativi (iperparametri settati dall'utente).

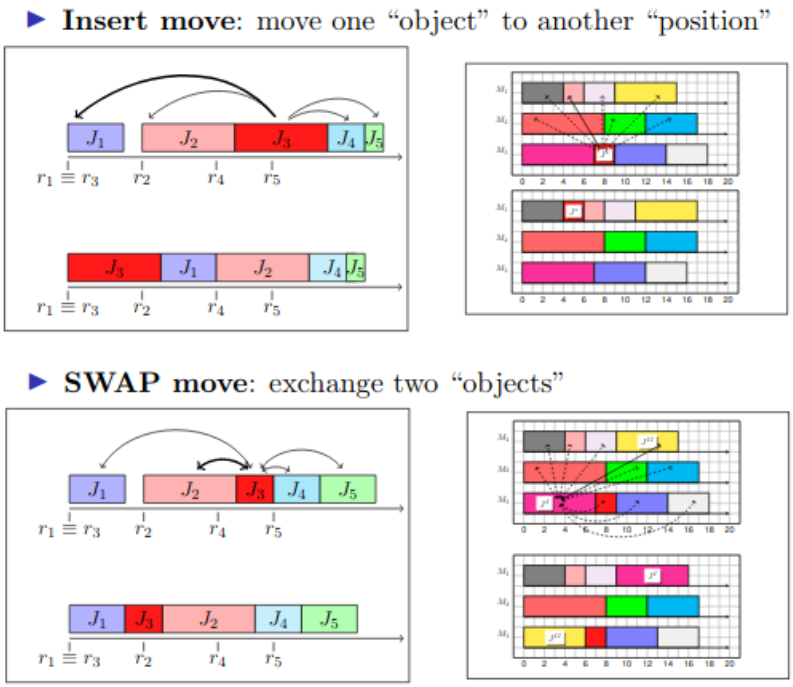


Figura 2: Neighborhood Search available moves

Algorithm 1: Heuristic

Input: n, m, w, p, N
Output: $\bar{S}$, schedule ottime selezionate

$\bar{S}, s \leftarrow \{\}$
$jobs \leftarrow$ smith_order(n, w, p)
$n\_iter \leftarrow 0$
while $n\_iter < N$ **do**
    $s \leftarrow$ create_rand_sched$(jobs)$
    $\bar{S} \leftarrow \bar{S} \cup s$
    n_iter++
$\bar{S} \leftarrow$ extract_best$(\bar{S}, 10)$
return neighborhood_search$(\bar{S})$

Da cui si può riassumere l'intero processo con il seguente **pseudocodice**:

Algorithm 2: Column Generation

Input: n, m, w, p

Output: x^*, schedule ottime selezionate

// inizializzare $\bar{S}$ con il metodo euristico introdotto in 2.3

$\bar{S} \leftarrow$ heuristic

repeat

  // risolvere rilassamento continuo sul set covering

  $\lambda, x^* \leftarrow$ column_generation($\bar{S}$)

  $F^* \leftarrow$ pricing(λ)

  if $F^* < 0$ **then**

    // aggiungere 3 nuove schedule 2.1.1

    $S' \leftarrow$ backtracing

    $\bar{S} \leftarrow \bar{S} \cup S'$

until $F^* < 0$

// ricavato x^* forzarne l'interezza

if $x^* \in [0, 1]$ ***and*** *Theorem 1* **then**

  $x^* \leftarrow$ simple_swap(x^*)

else if $x^* \in [0, 1]$ **then**

  $x^* \leftarrow$ branch&bound(x^*)

return x^*

3 Implementazione

Per implementare la soluzione è stato utilizzato il linguaggio di programmazione Python 3.6. Per la gestione dei dati si è utilizzata la libreria numpy versione 1.19.5, mentre per la risoluzione del modello LP si è utilizzato il risolutore di Gurobi, versione 9.1. Quest'ultima versione introduce la compatibilità con gli array numpy per la definizione di vincoli e funzione obiettivo; ovviamente è necessaria la licenza. Il codice e tutta la parte di documentazione è resa disponibile su github al link https://github.com/simoneCavana/CG_parallel-machine-sched.

3.1 Codice e il suo utilizzo

La struttura del codice si compone dei seguenti file:

- *config.toml*: file contentente i parametri di configurazione;
- *benchmark.py*: file di gestione delle istanze dei benchmark;
- *utilities.py*: file di gestione delle strutture dati e delle funzioni più utilizzate;
- *heuristic.py*: file che contiene l'implementazione dell' euristica;
- *optimization.py*: file che contiene l'implementazione di column generation compreso di pricing algorithm;
- *manager.py*: file contenente la gestione dell'albero e dei relativi nodi;
- *___main___.py*: file principale in cui viene controllato l'input dell'utente e viene lanciato l'algoritmo.

Inoltre sono presenti diverse cartelle che contengono ad esempio i benchmark di input, i risultati di output, il *.write()* dei modelli e la documentazione. In particolare i file di output sono nominati secondo la regola *tipo_m_n.txt* dove per tipo possiamo avere "beb" o "rnd" mentre m ed n rappresentano le macchine e il numero di jobs.

Inizialmente si era pensato di fare uso delle nuove variabili *Mvar* rese disponibili dalla versione di gurobi 9.0 che permettono una più facile implementazione delle operazioni matriciali ma, a seguito di diversi tentativi, non è stato possibile implementare l'append a questo tipo di oggetto perchè non permette di essere manipolato se non per recuperare gli elementi contenuti. Per questo motivo, dato che a seguito di Pricing bisogna inserire nuove variabili nel modello, si è tornati ad un implementazione classica con variabili di tipo Var.

Nell'applicazione dell'euristica per inizializzare la matrice a_{js} delle schedule sono stati realizzati due differenti approcci per la selezione delle *10m* migliori schedule. Il primo, un approccio naive, in cui si ordinano le soluzioni in ordine crescente e se ne selezionano le 10m migliori senza considerare le soluzioni identiche. Il secondo, più oneroso dal punto di vista computazionale, in cui oltre all'ordinamento crescente per costo si controlla che le soluzioni selezionate siano differenti le une dalle altre (nonostante questo alcune schedule possono comunque essere uguali ma non intere soluzioni). Questi due approcci portano ad un comportamento abbastanza differente durante l'esecuzione in quanto l'approccio naive risulta in un esecuzione molto rapida inizialmente ma conduce ad un maggior numero di esecuzioni, e qundi dispendio di tempo, il column generation. L'approccio più "tricky" invece porta ad un maggior dispendio di risorse nella parte di selezione delle schedule ma facilita il lavoro del column generation sucessivo in quanto si trova già con una varietà maggiore di schedule direttamente in $\bar{S}$.

La parte più onerosa dal punto di vista dell'implementazione è stata la gestione dell'albero di Branch & Bound e di tutti i nodi appartenenti ad esso. Come è possibile vedere dalle immagini presenti nella corrispettiva presentazione abbiamo che ogni nodo è strutturato come un dizionario le cui chiavi sono: *key, value, job, parent* e infine *children*. Tutte queste chiavi sono necessarie per la gestione dell'albero mentre per quanto riguarda la risoluzione del problema si ha che ogni nodo si fa carico di una lista di valori, contenuti in *value*, che servono a mantenere lo stato

del problema con i relativi oggetti. Infine, come è possibile immaginare, questi "nodi" sono mantenuti in una lista gestita a stack in cui appunto si fa push e pop dei nodi.

Siccome non è assicurato che tramite il Branch & Bound si trovi l'ottimo è stato scelto di implementare l'albero con un approccio *Depth First*, ovvero si va a cercare la prima soluzione ottima disponibile e la si ritorna.

3.2 Risultati

La macchina che è stata utilizzata per questi esperimenti è fornita di Intel© Core™ i5-6200U CPU @ 2.30GHz x 2, da cui il parallelismo di gurobi lavora quindi su quattro core virtuali. Per testare l'algoritmo implementato sono state utilizzate le 15 istanze fornite da Barnes e Brennan [4] come descritto in [3] e raffigurate in 3. Inoltre è stato realizzato un metodo di generazione random di istanze basato sull'implementazione descritta dall'articolo analizzato in cui vengono suddivise le istanze in tre categorie in base alla scelta della distribuzione dei valori:

1. processing time ricavati da una distribuzione normale di valori fra [1, 10] mentre i pesi dei job sono ricavati da una normale fra [10, 100];
2. sia i tempi d'esecuzione che i pesi dei lavori vengono estratti da una distribuzione normale fra [1, 100];
3. sia i tempi d'esecuzione che i pesi dei lavori vengono estratti da una distribuzione normale fra [1, 20].

Mentre m ed n vengono scelte rispettivamente dall'insieme di valori [3,4,5] e [20,30,40,50].

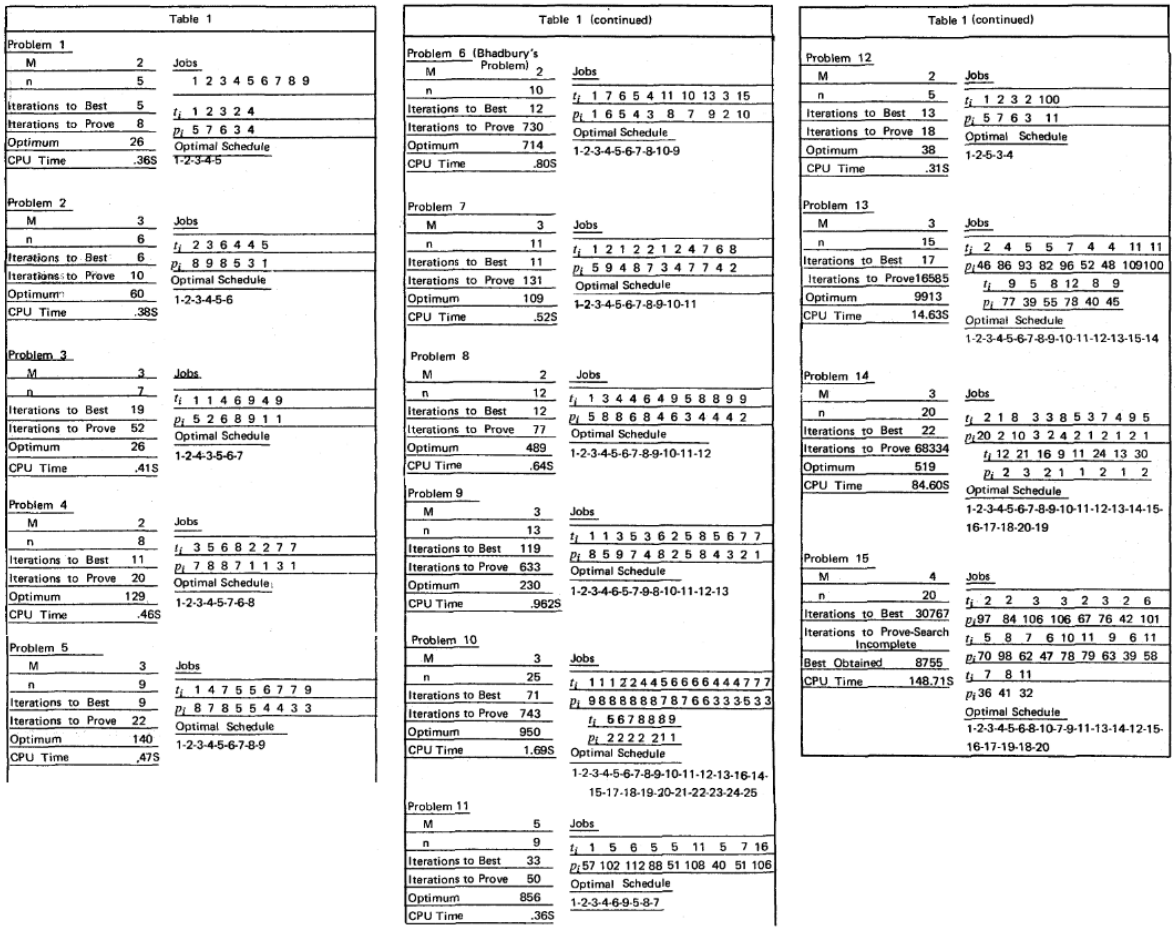


Table 1

Problem 1	
M	2
n	5
Jobs	1 2 3 4 5 6 7 8 9
Iterations to Best	5
Iterations to Prove	8
t_i	1 2 3 2 4
p_i	5 7 6 3 4
Optimum	26
Optimal Schedule	
CPU Time	.36S
	1-2-3-4-5
Problem 2	
M	3
n	6
Jobs	
t_i	2 3 6 4 4 5
Iterations to Best	6
p_i	8 9 8 5 3 1
Iterations to Prove	10
Optimum	60
Optimal Schedule	
CPU Time	.38S
	1-2-3-4-5-6
Problem 3	
M	3
n	7
Jobs	
t_i	1 1 4 6 9 4 9
Iterations to Best	19
p_i	5 2 6 8 9 1 1
Iterations to Prove	52
Optimal Schedule	
Optimum	26
	1-2-4-3-5-6-7
CPU Time	.41S
Problem 4	
M	2
n	8
Jobs	
t_i	3 5 6 8 2 2 7 7
Iterations to Best	11
p_i	7 8 8 7 1 1 3 1
Iterations to Prove	20
Optimal Schedule	
Optimum	129
	1-2-3-4-5-7-6-8
CPU Time	.46S
Problem 5	
M	3
n	9
Jobs	
t_i	1 4 7 5 5 6 7 7 9
Iterations to Best	9
p_i	8 7 8 5 5 4 4 3 3
Iterations to Prove	22
Optimal Schedule	
Optimum	140
	1-2-3-4-5-6-7-8-9
CPU Time	.47S

Table 1 (continued)

Problem 6 (Bhadbury's Problem)	
M	2
n	10
Jobs	
t_i	1 7 6 5 4 11 10 13 3 15
Iterations to Best	12
p_i	1 6 5 4 3 8 7 9 2 10
Iterations to Prove	730
Optimal Schedule	
Optimum	714
	1-2-3-4-5-6-7-8-10-9
CPU Time	.80S
Problem 7	
M	3
n	11
Jobs	
t_i	1 2 1 2 2 1 2 4 7 6 8
Iterations to Best	11
p_i	5 9 4 8 7 3 4 7 7 4 2
Iterations to Prove	131
Optimal Schedule	
Optimum	109
	1-2-3-4-5-6-7-8-9-10-11
CPU Time	.52S
Problem 8	
M	2
n	12
Jobs	
t_i	1 3 4 4 6 4 9 5 8 8 9 9
Iterations to Best	12
p_i	5 8 8 6 8 4 6 3 4 4 4 2
Iterations to Prove	77
Optimal Schedule	
Optimum	489
	1-2-3-4-5-6-7-8-9-10-11-12
CPU Time	.64S
Problem 9	
M	3
n	13
Jobs	
t_i	1 1 3 5 3 6 2 5 8 5 6 7 7
Iterations to Best	119
p_i	8 5 9 7 4 8 2 5 8 4 3 2 1
Iterations to Prove	633
Optimal Schedule	
Optimum	230
	1-2-3-4-6-5-7-9-8-10-11-12-13
CPU Time	.962S
Problem 10	
M	3
n	25
Jobs	
t_i	1 1 1 2 2 4 4 5 6 6 6 6 4 4 4 7 7 7
Iterations to Best	71
p_i	9 8 8 8 8 8 8 7 8 7 6 6 3 3 3 5 3 3
Iterations to Prove	743
t_i	5 6 7 8 8 8 9
Optimum	950
p_i	2 2 2 2 2 1 1
CPU Time	1.69S
Optimal Schedule	
	1-2-3-4-5-6-7-8-9-10-11-12-13-16-14-15-17-18-19-20-21-22-23-24-25
Problem 11	
M	5
n	9
Jobs	
t_i	1 5 6 5 5 11 5 7 16
Iterations to Best	33
p_i	57 102 112 88 51 108 40 51 106
Iterations to Prove	50
Optimal Schedule	
Optimum	856
	1-2-3-4-6-9-5-8-7
CPU Time	.36S

Table 1 (continued)

Problem 12	
M	2
n	5
Jobs	
t_i	1 2 3 2 100
Iterations to Best	13
p_i	5 7 6 3 11
Iterations to Prove	18
Optimal Schedule	
Optimum	38
	1-2-5-3-4
CPU Time	.31S
Problem 13	
M	3
n	15
Jobs	
t_i	2 4 5 5 7 4 4 11 11
Iterations to Best	17
p_i	46 86 93 82 96 52 48 109100
t_i	9 5 8 12 8 9
Iterations to Prove	16585
p_i	77 39 55 78 40 45
Optimum	9913
Optimal Schedule	
CPU Time	14.63S
	1-2-3-4-5-6-7-8-9-10-11-12-13-15-14
Problem 14	
M	3
n	20
Jobs	
t_i	2 1 8 3 3 8 5 3 7 4 9 5
Iterations to Best	22
p_i	20 2 10 3 2 4 2 1 2 1 2 1
t_i	12 21 16 9 11 24 13 30
Iterations to Prove	68334
p_i	2 3 2 1 1 2 1 2
Optimum	519
Optimal Schedule	
CPU Time	84.60S
	1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-20-19
Problem 15	
M	4
n	20
Jobs	
t_i	2 2 3 3 2 3 2 6
Iterations to Best	30767
p_i	97 84 106 106 67 76 42 101
t_i	5 8 7 6 10 11 9 6 11
Iterations to Prove-Search Incomplete	
p_i	70 98 62 47 78 79 63 39 58
t_i	7 8 11
Best Obtained	8755
p_i	36 41 32
CPU Time	148.71S
Optimal Schedule	
	1-2-3-4-5-6-8-10-7-9-11-13-14-12-15-16-17-19-18-20

Figura 3: Barnes and Brennan benchmark

Per la creazione della tabella 1 è stato fatto uso di pandas per semplificare l'analisi dei risultati presenti nei file *tabular_data.csv* nella cartella di output. Inoltre, è stato riscontrato che nella suddetta libreria è presente la funzione *DataFrame.to_latex()* realizzata ad hoc per esportare tabelle in formato Latex. Di seguito è riportata la definizione dell'intestazione delle tabelle:

m = numero di macchine;
n = numero di job;
$HTime$ = tempo d'esecuzione medio dell'euristica (in secondi);
$CGTime$ = tempo d'esecuzione medio di Column Generation (in secondi);
NB = numero di istanze sul totale in cui non è stato usato B&B;
MNN = numero massimo di nodi visitati;
$MGAP$ = gap massimo, $ILP - LB$;
ILP = numero di istanze sul totale in cui $LB = ILP$;
TOT = numero di istanze totali.

m	n	HTime	CGTime	NB	MNN	MGAP	ILP	TOT
2	5	0.75	0.00	57	0	0	57	57
	8	1.00	0.11	27	0	0	27	27
	10	1.73	2.33	27	1	0	33	33
	12	1.82	0.24	30	1	0	33	33
3	6	0.89	0.00	28	0	0	28	28
	7	0.65	0.00	17	0	0	17	17
	9	1.68	0.52	21	2	0	25	25
	11	1.76	0.18	15	2	0	17	17
	13	2.60	8.55	10	2	0	20	20
	15	2.76	0.34	29	0	0	29	29
	20	4.19	0.89	26	2	0	36	36
	25	5.83	0.46	20	3	0	24	24
4	20	4.06	0.75	32	0	0	32	32
5	9	1.93	0.07	30	0	0	30	30
3	20	4.03	0.22	35	1	0	36	36
	30	8.21	1.03	34	0	0	34	34
	40	17.62	3.14	29	0	0	29	29
	50	23.41	2.95	37	0	0	37	37
4	20	4.03	1.13	36	1	0	38	38
	30	9.60	0.87	29	1	4	29	30
	40	18.67	1.64	33	0	0	33	33
	50	27.00	2.58	43	0	0	43	43
5	20	3.56	1.50	33	1	2	35	36
	30	8.95	4.40	14	7	16	18	20
	40	20.89	3.61	25	1	0	28	28
	50	27.43	2.45	44	0	0	44	44

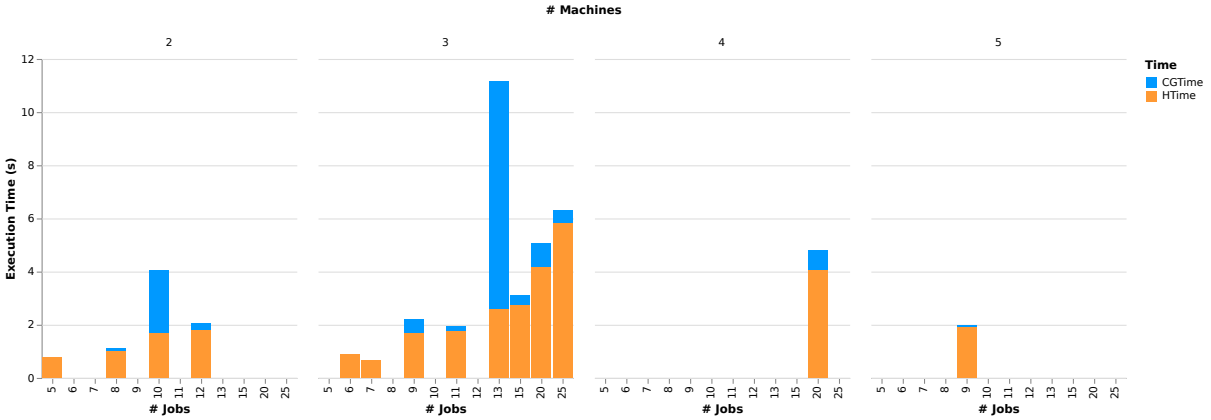
Tabella 1: Comparison table

La tabella è stata rappresentata volutamente divisa in due parti per suddividere i dati ricavati dall'esecuzione sui due principali benchmark presentati. Oltre a questo si è scelto di non ripetere i valori nella colonna m delle macchine per evitare ridondanza aiutandone la lettura tramite i colori. Per quanto riguarda l'analisi di questa tabella mi sono munito, sempre attraverso pandas, di grafici per comprendere in maniera più efficace e semplice i dati. Come si può vedere dalle immagini in 4 i tempi d'esecuzione sono caratterizzati soprattuto dal tempo di processing

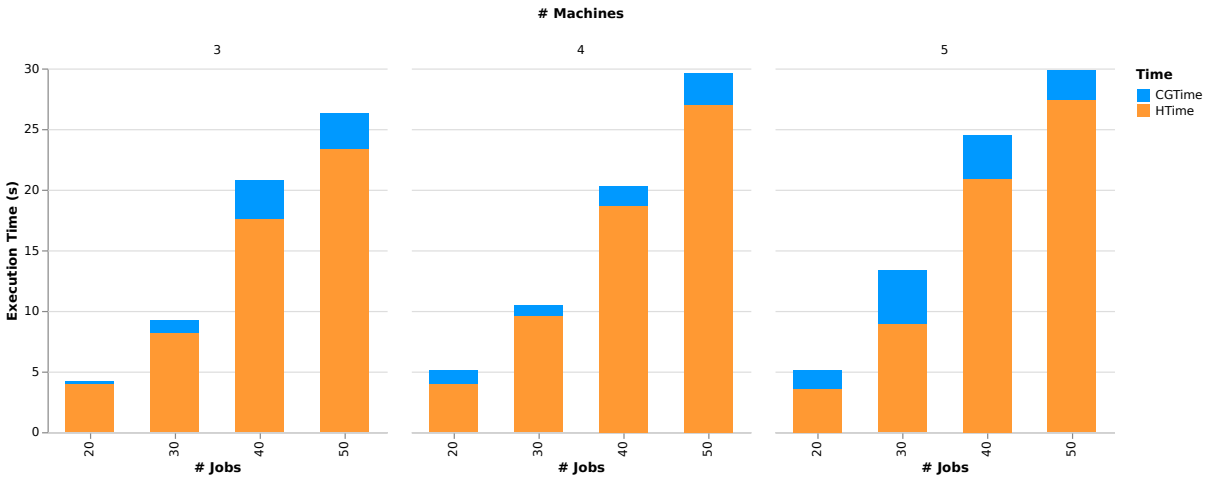
dell'euristica, in quanto per la parte di Column Generation abbiamo quasi sempre tempi irrisori se non per quei casi in cui si deve fare Branch & Bound in cui appunto il tempo d'esecuzione anche per questa parte aumenta considerevolmente.

Infine possiamo vedere che nella maggior parte delle iterazioni non è necessario l'utilizzo di branch & bound e, per quanto riguarda le istanze di Barnes & Brennan, quando è richiesto viene sfruttato su pochi nodi prima di trovare la soluzione ottima pari al Lower Buond (vedi colonna ILP). Volgendo questo ragionamento alle istanze generate casualmente si ha che anche qui in pochi casi si è dovuto iterare su diversi nodi dell'albero prima di trovare una soluzione spesso pari al lower bound.

Volendo paragonare i risultati ottenuti con quelli dell'articolo si nota subito che i tempi d'esecuzione sono molto ridotti, questo molto probabilmente è dovuto al fatto che i risultati dell'articolo risalgono al '99 ed erano eseguiti su una macchina ad oggi obsoleta, nonostante il codice fosse scritto in linguaggio C. Nel nostro caso inoltre non si trova più la discrepanza data dal fatto che a parità di job, con l'aumentare del numero di macchine le performance aumentano. Quest'ultima caratteristica può essere invece che sia stata limata dal fatto che non ho mantenuto una suddivisione sulle istanze dei benchmark random come fatto nell'articolo. Però si può notare la caratteristica per cui fissato $n = 30$ ed m variabile è il caso più compicato e in cui solitamente è richiesto Branch & Bound.



(a) Mean stacked computational time on Barnes & Brennan benchmark



(b) Mean stacked computational time on Random benchmark

Figura 4: Comparing Heuristic and Column Generation execution time for increasing jobs grouped on machines

4 Conclusioni

Si può concludere che, nonostante le difficoltà implementative di Branch & Bound, si è riusciti a realizzare con successo quanto descritto nell'articolo. Sono state poste delle scelte cruciali per quelle situazioni in cui non era ben specificato come comportarsi e l'algoritmo funziona in modo alquanto efficiente. Purtroppo vi sono ancora casistiche in cui il B&B o il column generation impiegano troppo tempo. Tuttavia, l'implementazione lascia spazio ancora a diverse ottimizzazioni come ad esempio quelle proposte in appendice [3] sull'algoritmo di pricing. Si potrebbe inoltre pensare di voler implementare come sviluppo futuro il metodo di Branch & Bound proposto da Chen et al. [2] in modo da paragonare i tempi di esecuzione su richiesta di B&B. Quest'ultima aggiunta in realtà servirebbe più a scopo di analisi pratica della descrizione teorica fornita in quanto è già descritto che andrebbe a modificare e quindi ad aggiungere un $\mathcal{O}(n)$ alla computazione totale.

Riferimenti bibliografici

[1] Marjan van den Akker, Han Hoogeveen, and Steef van de Velde. *Applying Column Generation to Machine Scheduling*, pages 303–330. Springer US, Boston, MA, 2005.

[2] Zhi-Long Chen and Warren B. Powell. Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing*, 11(1):78–94, 1999.

[3] Marjan van den Akker, Han Hoogeveen, and Steef van de Velde. Parallel machine scheduling by column generation. *Operations Research*, 47(6):862–872, 1999.

[4] J. Wesley Barnes and J. J. Brennan. An improved algorithm for scheduling jobs on identical machines. *AIIE Transactions*, 9(1):25–31, 1977.