دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر
استاد درس: دکتر مهدی دهقان
بهار ۱۴۰۲

موازی سازی الگوریتم تجزیه QR

علی عبدالهیان نوقابی

جبر خطی عددی

فهرست مطالب

۱ چکیده

جبر خطی عددی نقش مهمی در محاسبات علمی ایفا می‌کند و ابزارها و تکنیک‌های اساسی را برای حل طیف گسترده‌ای از مسائل ریاضی فراهم می‌سازد. تجزیه ماتریس، جنبه‌ای اساسی از جبر عددی است که امکان ساده‌سازی عملیات پیچیده ماتریسی را فراهم می‌کند و به حل سیستم‌های خطی، مسائل مقادیر ویژه و محاسبه معکوس‌های ماتریس کمک می‌کند. الگوریتم‌های علمی که بر این پایه‌های جبری بنا شده‌اند، در زمینه‌های مختلفی از جمله فیزیک، مهندسی، علوم کامپیوتر و اقتصاد، که در آن‌ها از شبیه‌سازی‌ها، بهینه‌سازی‌ها و وظایف تحلیل داده‌ها حمایت می‌کنند، نقش اساسی دارند.

تکامل سخت‌افزارهای محاسباتی و رشد نمایی اندازه‌های داده، نیاز به افزایش کارایی الگوریتم‌های علمی را برجسته کرده است. این نیاز به توسعه و پذیرش تکنیک‌های محاسبات موازی، که وظایف محاسباتی را در میان چندین پردازنده توزیع می‌کنند، منجر شده است. با موازی‌سازی الگوریتم‌ها، کاهش قابل توجهی در زمان اجرای آن‌ها می‌توان به دست آورد و امکان مواجهه با مسائل بزرگ‌مقیاسی که قبلاً غیرقابل حل بودند فراهم می‌شود.

الگوریتم‌های تجزیه ماتریس، مانند تجزیه QR، به دلیل شدت محاسباتی و استفاده مکرر در برنامه‌های علمی و مهندسی، کاندیداهای اصلی برای موازی‌سازی هستند. تجزیه QR به‌طور خاص، یک ماتریس را به یک ماتریس متعامد (Q) و یک ماتریس مثلثی بالایی (R) تجزیه می‌کند و یک سنگ بنای اساسی برای حل مسائل حداقل مربعات خطی و محاسبات مقادیر ویژه است. با این حال، پیاده‌سازی‌های سنتی دنباله‌ای این الگوریتم‌ها برای برآورده کردن نیازهای محیط‌های محاسباتی با کارایی بالا کافی نیستند.

پیاده‌سازی‌های موازی تجزیه QR، با بهره‌گیری از تکنیک‌هایی مانند تبدیل هاوس‌هولدر و الگوریتم گرام‌ـاشمیت اصلاح‌شده، بهبودهای قابل توجهی در عملکرد ارائه می‌دهند. این الگوریتم‌ها می‌توانند بر روی معماری‌های موازی اجرا شوند، از جمله سیستم‌های گذر پیام که در آن ماتریس به بلوک‌های ردیفی در میان پردازنده‌ها توزیع می‌شود. علیرغم چالش‌های مرتبط با سربار ارتباطی و همزمان‌سازی، الگوریتم‌های موازی متوسط‌ـدانه کارآیی خود را در متوازن‌سازی محاسبات و ارتباطات نشان داده‌اند و بدین ترتیب عملکرد را بهینه می‌کنند.

به طور خلاصه، نقش حیاتی جبر عددی و تجزیه ماتریس در محاسبات علمی، همراه با ضرورت موازی‌سازی الگوریتم‌ها، پیشرفت‌های مداوم در این حوزه را برجسته می‌سازد. همانطور که مسائل محاسباتی پیچیده‌تر و بزرگ‌تر می‌شوند، توسعه الگوریتم‌های موازی کارآمد برای پیشرفت کشف‌های علمی و نوآوری‌ها ضروری خواهد بود.

در این پروژه، پیاده‌سازی موازی دو الگوریتم برای انجام تجزیه QR یک ماتریس انجام شده است. بر اساس نتایج آزمایش ها الگوریتم‌های موازی برای گرام‌ـاشمیت اصلاح شده و الگوریتم‌های هاوس‌هولدر را بر روی سیستم‌های ارسال پیام عملکرد بهتری دارد که در آن ماتریس به صورت بلوک‌های سطری توزیع می‌شود.

کلیدواژه‌ها: تجزیه QR، الگوریتم گرام‌ـاشمیت، الگوریتم هاوس‌هولدر، سیستم‌های ارسال پیام.

۲ مقدمه

۱.۲ تجزیه QR

تجزیه QR یک روش در جبر خطی است که برای تجزیه یک ماتریس مستطیلی A با ابعاد $m \times n$ به دو ماتریس Q و R استفاده می‌شود که:

$$A = QR$$

در اینجا:

- Q ماتریسی متعامد $m \times m$ است که $Q^TQ = I$، که I ماتریس همانی است.
- R یک ماتریس بالا مثلثی $m \times n$ است.

این تجزیه کاربردهای فراوانی دارد از جمله در حل دستگاه‌های معادلات خطی، محاسبه بردارهای ویژه و مقادیر ویژه، و تحلیل مؤلفه‌های اصلی. یکی از مزایای اصلی استفاده از تجزیه QR برای محاسبه بردارهای ویژه این است که بر خلاف تجزیه LU، نیاز به مربع بودن ماتریس ندارد.

۲.۲ الگوریتم گرام‌ـاشمیت

الگوریتم گرام‌ـاشمیت [1] یک روش کلاسیک برای متعامد و یکه سازی مجموعه‌ای از بردارها در فضاهای برداری است. فرم ریاضی این الگوریتم به صورت زیر است:

۱. فرض کنید $v_1, v_2, \ldots, v_n$ بردارهای اولیه باشند.

۲. برای $i = 1$ تا n:

$$u_i = v_i - \sum_{j=1}^{i-1} \frac{\langle u_j, v_i \rangle}{\langle u_j, u_j \rangle} u_j$$

$$e_i = \frac{u_i}{\|u_i\|}$$

که در آن $\langle \cdot, \cdot \rangle$ نشان‌دهنده ضرب داخلی است و e_iها بردارهای متعامد و یکه به دست آمده هستند.

۳.۲ ماتریس‌های هاوس‌هولدر

ماتریس‌های هاوس‌هولدر [2] برای تولید تبدیلات انعکاسی استفاده می‌شوند که در تجزیه‌های ماتریسی و حل معادلات خطی کاربرد دارند. این تبدیلات به خصوص برای ایجاد صفرهای زیر قطری در ماتریس‌ها استفاده می‌شوند و به این ترتیب، ماتریس‌ها را به شکل مثلثی یا هسنبرگ [3] تبدیل می‌کنند. فرمول ریاضی این ماتریس به صورت زیر است:

$$H = I - 2\frac{vv^T}{v^Tv}$$

که v یک بردار نرمال‌سازی نشده است و H ماتریس انعکاسی متعامدی است که بردار x را به برداری در جهت محورهای اصلی تبدیل می‌کند. این تبدیل معمولاً برای ساده‌سازی محاسبات در الگوریتم‌های پیچیده‌تر مانند تجزیه QR استفاده می‌شود.

[1] Gram-Schmidt algorithm
[2] Householder matrices
[3] Hessenberg

در استفاده از ماتریس‌های هاوس‌هولدر برای صفر کردن عناصر زیر قطری، ابتدا بردار v را به گونه‌ای انتخاب می‌کنیم که تبدیل هاوس‌هولدر H، زمانی که به ماتریس A اعمال شود، عناصر زیر قطری را در ستون‌های خاص صفر کند. به این صورت که:

$$v = x + \text{sign}(x_1)\|x\|e_1$$

که در آن x ستونی از ماتریس A است که می‌خواهیم آن را تبدیل کنیم، x_1 اولین عنصر x، $\|x\|$ نرم یوکلیدی x و e_1 اولین بردار پایه استاندارد است. این انتخاب از v باعث می‌شود که Hx یک بردار در جهت e_1 باشد و بنابراین عناصر زیر قطر در x صفر می‌شوند.

۴.۲ رویکرد های مختلف در موازی سازی

۱.۴.۲ سیستم‌های ارسال پیام

سیستم‌های ارسال پیام[۴] معماری کامپیوتری است که در آن پردازنده‌ها با ارسال پیام به یکدیگر برای هماهنگی و اجرای دستورات ارتباط برقرار می‌کنند. این روش میتواند هزینه‌های ارتباطی بالا داشته باشد که بر سرعت برنامه‌ها تأثیر منفی بگذارد، به‌خصوص اگر نیاز به تبادل پیام‌های مکرر باشد. این روش در مدل‌سازی‌های آب و هوایی بر روی سوپرکامپیوترها که در آنها گره‌ها داده‌های مربوط به مناطق جغرافیایی مختلف را پردازش می‌کنند استفاده میشود.

۲.۴.۲ رویکرد سراسری

رویکرد سراسری [۵] شروع به تحلیل یک الگوریتم ترتیبی می‌کند و بخش‌هایی که قابلیت اجرای موازی دارند را انجام میدهد. این رویکرد می‌تواند به افزایش سرعت برنامه‌ها کمک کند، اما اثربخشی آن به قابلیت‌های موازی‌سازی الگوریتم اولیه بستگی دارد. در تبدیل نرم‌افزارهای معمولی به نسخه‌هایی که بر روی سیستم‌های موازی اجرا می‌شوند استفاده میشود، مانند پردازش تصاویر یا پایگاه‌داده‌ها.

۳.۴.۲ رویکرد موضعی

در رویکرد موضعی[۶]، الگوریتم‌ها از ابتدا برای عملیات موازی طراحی می‌شوند و بر تبدیل‌های مستقل روی عناصر یا بلوک‌های داده تمرکز دارند. این رویکرد با کاهش وابستگی‌های داده‌ای غیر ضروری و ارتباطات، می‌تواند کارایی را به حداکثر برساند. در محاسبات عملکرد بالا مانند عملیات روی ماتریس‌ها و سایر روش‌های عددی که نیاز به دستکاری مفصل داده‌ها دارند استفاده میشود.

۴.۴.۲ الگوریتم‌های دانه‌ریز

الگوریتم‌های دانه‌ریز [۷] مسئله را به بخش‌های کوچک‌تری تقسیم می‌کنند که هر کدام به طور همزمان پردازش می‌شوند. اگرچه این الگوریتم‌ها می‌توانند موازی‌سازی بالایی را ارائه دهند، ولی به شدت به هزینه‌های همگام‌سازی [۸]و ارتباط بستگی دارند. مناسب برای سیستم‌هایی با هزینه ارتباطی پایین و توانایی همگام‌سازی بالا مانند پردازنده‌های چند هسته‌ای با حافظه مشترک است.

[4] Message Passing Systems
[5] Global Approach
[6] Local Approach
[7] Fine-Grained Algorithms
[8] overhead of synchronization

۵.۴.۲ الگوریتم‌های دانه‌متوسط

الگوریتم‌های دانه‌متوسط [9] بین رویکردهای دانه‌ریز و درشت‌دانه تعادل برقرار می‌کنند و وظایف کمتر و بزرگتری را نسبت به دانه‌ریز ایجاد می‌کنند. با کاهش نیاز به ارتباط مکرر، این الگوریتم‌ها می‌توانند عملکرد بهتری در سیستم‌هایی که هزینه ارتباطی نسبتاً بالا دارند ارائه دهند. مناسب برای سیستم‌های توزیع شده که در آن هزینه‌های ارتباطی چشمگیر است، مانند شبکه‌های کلاستر یا محاسبات گرید است.

۶.۴.۲ الگوریتم‌های درشت‌دانه

الگوریتم‌های درشت‌دانه [10] مسئله را به زیر وظایف نسبتاً بزرگ تقسیم می‌کنند، که هر کدام شامل مقدار زیادی محاسبه قبل از هر گونه ارتباط یا همگام‌سازی می‌باشد. این الگوریتم‌ها با کاهش فرکانس و هزینه ارتباط، می‌توانند در محیط‌هایی با هزینه ارتباطی بالا به خوبی کار کنند. مناسب برای سیستم‌های توزیع شده بزرگ مقیاس مانند محاسبات ابری که در آن گره‌های محاسباتی جغرافیایاً پراکنده‌اند است.

۳ مطالعات مرتبط

الگوریتم‌های تجزیه QR در ماشین‌های پردازش موازی در بسیاری از زمینه‌ها مورد مطالعه قرار گرفته‌اند. یک بررسی در [۷] ارائه شده است. الگوریتم‌های سیستولیک مبتنی بر چرخش‌های گیونز [11] در بارلو و ایپسن [۲]، بوجانچیک [۴]، هلر و ایپسن [۱۲]، ایپسن [۱۳]، و لوک [۱۶] ارائه شده است. الگوریتم‌های گیونز بر روی ماشین‌های حافظه اشتراکی در سامه و کک [۲۶]، کزنار، مولر و رابرت [۸]، دونگارا، سامه و سورنسن [۹]، مودی و کلارک [۱۷] و لرد [۱۵] مطالعه شده است. الگوریتم‌های گیونز برای ماشین‌های حافظه توزیع شده در الدن [۱۰] برای مش‌های یک بعدی و دو بعدی و توروس، پوتن و راگاوان [۲۳] برای ارتباطات حلقه و پخش، و در چمبرلین و پاول [۵]، پوتن، سومش و وملپاتی [۲۴]، و چو و جرج [۶] برای هایپرکیوب ارائه شده‌اند. الگوریتم‌های مبتنی بر بازتاب‌های هاوسهولدر در دونگارا، سامه و سورنسن [۹] و کاتولی و سوتر [۱۴] برای حافظه‌های اشتراکی، در الدن [۱۰] برای مش‌های یک بعدی و دو بعدی و توروس، و در پوتن و راگاوان [۲۳] برای تقسیم‌بندی ستون با ارتباطات حلقه یا پخش ارائه شده‌اند. به نظر می‌رسد که پیاده‌سازی موازی الگوریتم گرام‌ـ اشمیت اصلاح‌شده مورد مطالعه قرار نگرفته است.

در بخش های ۴ و ۵ به بررسی موازی سازی الگوریتم گرام‌ـاشمیت و هوس هولدر می پردازیم. در بخش ۶ نتایج عددی آزمایش های انجام شده گردآوری و تحلیل شده. بخش های عمده ای از این گزارش بر اساس مقاله [۱] نوشته شده است.

[9] Medium-Grained Algorithms
[10] Coarse-Grained Algorithms
[11] Givens rotation

۴ موازی سازی الگوریتم گرام‌ـ اشمیت

الگوریتم گرام‌ـ اشمیت موازی از سیستم‌های ارسال پیام برای پیاده‌سازی مراحل الگوریتم گرام‌ـ اشمیت به صورت موازی بهره می‌برد. این فرایند به منظور بهبود کارایی و کاهش زمان اجرا در تجزیه ماتریس‌ها در محیط‌های با عملکرد بالا طراحی شده است.

۱.۴ رویکرد موازی‌سازی

۱.۱.۴ محاسبه ضرب‌های داخلی

هر پردازنده ضرب داخلی[۱۲] بلوک خود از ستون i را با تمام ستون‌های بعدی $i+1$ تا n محاسبه می‌کند. این کار به معنای تعیین میزان تأثیر هر ستون بر روی ستون‌های بعدی است و برای محاسبه اجزاء R مورد نیاز است.

$$\text{dotprd}[i] = A[:, i]^T \cdot A[:, i+1 : n]$$

۲.۱.۴ تجمع نتایج

پس از محاسبه ضرب‌های داخلی، پردازنده‌ها این ضرب‌ها را از پردازنده‌های دیگر دریافت می‌کنند و آن‌ها را با نتایج خود تجمیع[۱۳] می‌کنند تا یک مجموع کامل برای تمام پردازنده‌ها بدست آید. این امر برای اطمینان از دقت در محاسبات بعدی ضروری است.

$$\text{accum}[i] = \sum_{p=1}^{P} \text{dotprd}[i]_p$$

۳.۱.۴ انتقال داده‌ها

پردازنده‌ها نتایج تجمعی ضرب‌های داخلی را به پردازنده بعدی در حلقه ارسال می‌کنند. این انتقال[۱۴] برای همگام‌سازی داده‌ها در تمام پردازنده‌ها و آماده‌سازی برای مراحل بعدی ضروری است.

$$\text{pass}[i] = \text{processor next to send}(\text{accum}[i])$$

۴.۱.۴ مقیاس‌بندی

پردازنده مسئول، نرم ستون i را محاسبه می‌کند و سپس تمام ستون‌های i را به این نرم مقیاس[۱۵] می‌کند تا ستون نرمالیز شود. این نرمال‌سازی بخشی از فرایند تولید ماتریس Q است.

$$\text{scale}[i] = \frac{A[:, i]}{\|A[:, i]\|}$$

[12] Dot Products
[13] Accumulation
[14] Pass
[15] Scaling

۵.۱.۴ پخش کردن

پردازنده‌ها داده‌های مقیاس‌بندی شده و به‌روزرسانی‌های دیگر را برای استفاده توسط تمام پردازنده‌ها در مراحل بعدی پخش[۱۶] می‌کنند. این اقدام اطمینان می‌دهد که تمام پردازنده‌ها دارای داده‌های به‌روز هستند.

$$\text{brdcast}[i] = \text{processors all to broadcast}(A[:, i])$$

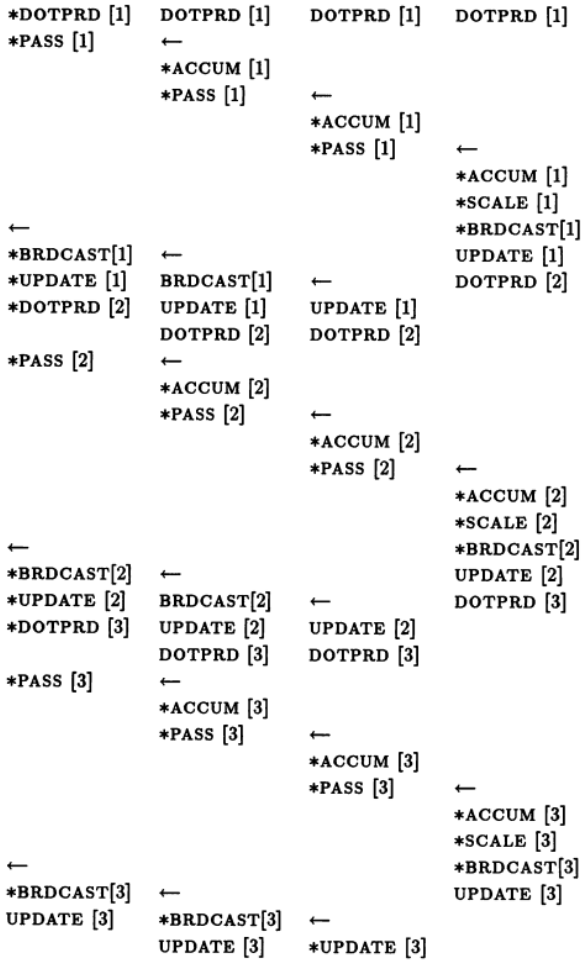
۶.۱.۴ به‌روزرسانی

هر پردازنده بلوک‌های خود از ستون‌های $i+1$ تا n را بر اساس ستون i نرمال شده به‌روز[۱۷] می‌کند. این کار شامل کم کردن مؤلفه‌های ستون i از ستون‌های بعدی برای حفظ متعامد بودن است.

$$A[:, i+1:n] = A[:, i+1:n] - (\text{dotprd}[i] \times A[:, i])$$

[16] Broadcast
[17] Update

۲.۴ دیاگرام زمان‌بندی



شکل ۱: نمودار زمان‌بندی برای الگوریتم گرام_اشمیت موازی

در دیاگرام شکل ۱، هر ستون نمایانگر فعالیت‌های یک پردازنده در حلقه است و هر سطر نشان‌دهنده گام‌های زمانی متوازی است. فلش‌ها وابستگی‌های بین عملیات در پردازنده‌های مختلف را نشان می‌دهند، که بیانگر نیاز به داده‌های پردازش شده توسط پردازنده‌های دیگر پیش از ادامه فعالیت است.

۱.۲.۴ توضیح نمودار

این فرآیندها نشان‌دهنده چگونگی همکاری پردازنده‌ها در محاسبه تجزیه QR به صورت موازی است و تأکید بر اهمیت دقیق بودن زمان‌بندی و هماهنگی بین پردازنده‌ها در شبکه حلقه‌ای دارد.

- **DOTPRD [۱]**: در این مرحله، پردازنده اول ضرب داخلی ستون اول ماتریس A را با خود و ستون‌های بعدی محاسبه می‌کند. این کار مقدمات متعامدسازی ستون‌های بعدی را فراهم می‌آورد.
- **PASS [۱]** و **ACCUM [۱]**: پردازنده اول نتایج ضرب داخلی را به پردازنده دوم می‌فرستد. پردازنده دوم این داده‌ها را با نتایج محاسبه شده خود تجمیع می‌کند.
- **UPDATE [۱]**: پس از آن که تمام پردازنده‌ها ضرب داخلی کامل را دریافت کرده‌اند، هر پردازنده ستون‌های خود را بر اساس نتایج ضرب داخلی به‌روزرسانی می‌کند تا اطمینان حاصل شود که هر ستون به ستون‌های قبل از خود متعامد باشد.
- **BRDCAST [۱]**: در این مرحله، پردازنده‌ای که به‌روزرسانی نهایی ستون اول را انجام داده است، نتایج را برای استفاده در گام‌های بعدی به سایر پردازنده‌ها پخش می‌کند.

۳.۴ تحلیل زمانی برای محیط‌های محاسباتی موازی

در اجرای موازی الگوریتم گرام‌ـ اشمیت بر روی یک حلقه از پردازنده‌ها، هر پردازنده الگوی خاصی از اجرا را دنبال می‌کند که شامل ارسال داده‌ها (برودکست)، به‌روزرسانی و محاسبه ضرب‌های داخلی برای ستون بعدی است. این الگو به دلیل وابستگی‌های بین پردازنده‌ها با تأخیرهایی همراه است.

$$\mathrm{TIME}_{\mathrm{GS,ring}} = \sum_{i=1}^{n} (\mathrm{DOTPRD}[i] + (p-1)\mathrm{ACCUM}[i] + \mathrm{SCALE}[i])$$

$$+(p-1)\mathrm{PASS}[i] + 2\mathrm{BRDCAST}[i] + \mathrm{UPDATE}[i] + (p-3)\mathrm{BRDCAST}[n].$$

[۱۸] **معرفی معماری هایپرکیوب:** معماری هایپرکیوب یکی از ساختارهای موثر برای شبکه‌های محاسباتی موازی است. هایپرکیوب امکان دسترسی سریع‌تر و کارآمدتر به داده‌ها را فراهم می‌آورد و تأخیرهای شبکه را به دلیل تعداد لینک‌های کمتر بین پردازنده‌ها کاهش می‌دهد. این معماری به خصوص در محاسبات پیچیده و بزرگ‌مقیاس که نیازمند همگام‌سازی و تبادل گسترده داده‌ها است، مفید است.

برای شبکه‌هایی با معماری هایپرکیوب:

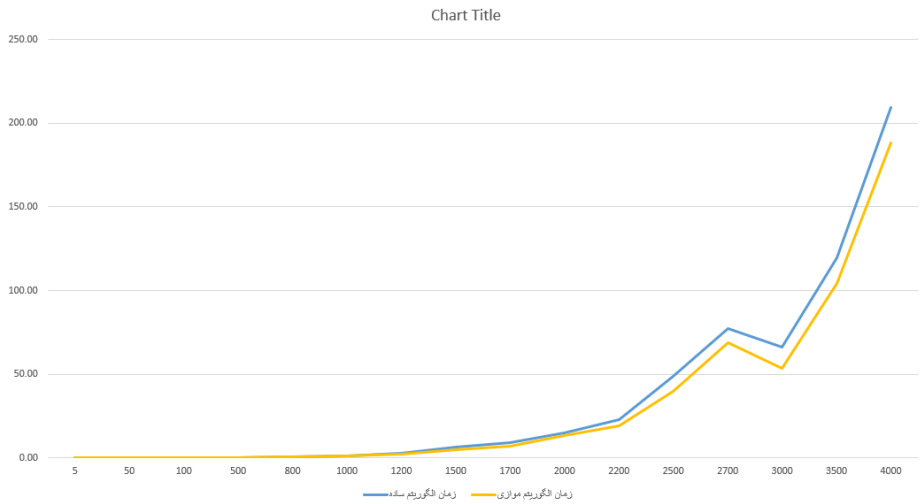
$$\mathrm{TIME}_{\mathrm{GS,cube}} = \sum_{i=1}^{n} (\mathrm{DOTPRD}[i] + (\log_2 p - 1)\mathrm{ACCUM}[i] + \mathrm{SCALE}[i])$$

$$+ \log_2 p\mathrm{PASS}\,[i] + \log_2 p\mathrm{BRDCAST}\,[i] + \mathrm{UPDATE}[i].$$

تعداد بهینه پردازنده‌ها برای حداقل کردن زمان اجرا با توجه به تعداد سطرها محاسبه می‌شود و تابعی از جذر تعداد سطرها است. تحلیل‌های زمانی دقیق‌تر می‌تواند به تعیین بهترین توزیع بار و کاهش زمان اجرای کلی کمک کند.

[18] hypercube

۴.۴ نمودار مقایسه الگوریتم ساده و الگوریتم موازی



شکل ۲: نمودار مقایسه زمان اجرای الگوریتم گرام_اشمیت سریال و موازی بر اساس اندازه ماتریس.

در این نمودار، زمان پردازش الگوریتم گرام_اشمیت به صورت سری و موازی برای ماتریس‌های مختلف نمایش داده شده است. همانطور که مشاهده می‌شود، با افزایش اندازه ماتریس، زمان الگوریتم موازی بهتر می‌شود.[۲۷]

۵.۴ پیاده‌سازی به زبان سی پلاس پلاس

کد پیاده‌سازی شده برای الگوریتم گرام_اشمیت موازی، از کتابخانه MPI برای توزیع محاسبات و داده‌ها بین پردازنده‌ها در یک محیط موازی استفاده می‌کند. این روش امکان بهره‌برداری از توان پردازشی چندین واحد پردازشی را فراهم می‌آورد و در نتیجه کاهش قابل توجهی در زمان اجرای کلی الگوریتم دارد.[۲۷]

```
void parallelGramSchmidt(const std::vector<std::vector<double>> &local_A,
 std::vector<std::vector<double>> &local_Q, std::vector<std::vector<double>> &R, int cols, int local_rows) {
    local_Q = local_A;
    R.resize(cols, std::vector<double>(cols, 0.0));

    for (int k = 0; k < cols; ++k) {
        double local_norm = 0.0;
        for (int i = 0; i < local_rows; ++i) {
            local_norm += local_Q[i][k] * local_Q[i][k];
        }

        double global_norm;
        MPI_Allreduce(&local_norm, &global_norm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        global_norm = sqrt(global_norm);

        for (int i = 0; i < local_rows; ++i) {
            local_Q[i][k] /= global_norm;
        }

        R[k][k] = global_norm;

        for (int j = k + 1; j < cols; ++j) {
            double local_dot = 0.0;
            for (int i = 0; i < local_rows; ++i) {
                local_dot += local_Q[i][k] * local_Q[i][j];
            }

            double global_dot;
            MPI_Allreduce(&local_dot, &global_dot, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

            R[k][j] = global_dot;

            for (int i = 0; i < local_rows; ++i) {
                local_Q[i][j] -= global_dot * local_Q[i][k];
            }
        }
    }
}

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::vector<std::vector<double>> A, local_A, local_Q, R;
    int rows, cols;

    MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);

    int local_rows = rows / size;
    local_A.resize(local_rows, std::vector<double>(cols));

    for (int i = 0; i < cols; ++i) {
        std::vector<double> temp(rows);
        if (rank == 0) {
            for (int j = 0; j < rows; ++j) {
                temp[j] = A[j][i];
            }
        }
        MPI_Scatter(temp.data(), local_rows, MPI_DOUBLE, &local_A[0][i], local_rows, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    parallelGramSchmidt(local_A, local_Q, R, cols, local_rows);

    // Gather results from all processes
    std::vector<std::vector<double>> global_Q(rows, std::vector<double>(cols));
    for (int i = 0; i < cols; ++i)
    {
        MPI_Gather(&local_Q[0][i], local_rows, MPI_DOUBLE, &global_Q[0][i], local_rows, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
}
```

۵ موازی سازی الگوریتم هاوس‌هولدر

الگوریتم هاوس‌هولدر موازی به منظور بهینه‌سازی تجزیه QR با استفاده از سیستم‌های ارسال پیام طراحی شده است. این الگوریتم به‌طور خاص به ساختار ماتریس A به شکل بالا مثلثی R و ماتریس متعامد Q می‌پردازد. الگوریتم هاوس‌هولدر با تبدیل ماتریس A به شکل بالا مثلثی، یک ستون در هر مرحله را کاهش می‌دهد. این تبدیل با استفاده از عملگرهای هاوس‌هولدر انجام می‌شود که هر ستون را به صورت تکراری به فرم مورد نیاز تبدیل می‌کند.

۱.۵ مراحل تبدیل هاوس‌هولدر

در مرحله i ام، الگوریتم یک بردار انعکاس[۱۹] u_i را تعریف می‌کند تا ستون i از ماتریس A را به یک بردار با مؤلفه‌های صفر در زیر قطر تبدیل کند. بردار u_i با استفاده از فرمول زیر محاسبه می‌شود:

$$u_i = \text{sign}(a_{ii}) \cdot \|a_{i:}\|_2 \cdot e_i + a_{i:}$$

$$\beta_i = -\frac{2}{u_i^T u_i}$$

که در آن $a_{i:}$ ستون i از ماتریس A است، e_i بردار یکه است که تنها در مؤلفه iام یک و در سایر مؤلفه‌ها صفر است.

با استفاده از بردار u_i، ماتریس انعکاس[۲۰] H_i به صورت زیر ساخته می‌شود:

$$H_i = I - \beta_i u_i u_i^T$$

که I ماتریس همانی است. ماتریس H_i برای تبدیل ستون i و ستون‌های بعدی A به کار می‌رود:

$$A \leftarrow H_i A$$

این فرآیند تا زمانی که تمام ستون‌های A به شکل بالا مثلثی تبدیل شوند تکرار می‌شود. در نهایت، ماتریس A به R تبدیل شده و ماتریس Q به عنوان حاصل ضرب معکوس‌های H_i محاسبه می‌شود:

$$Q = H_1^T H_2^T \dots H_n^T$$

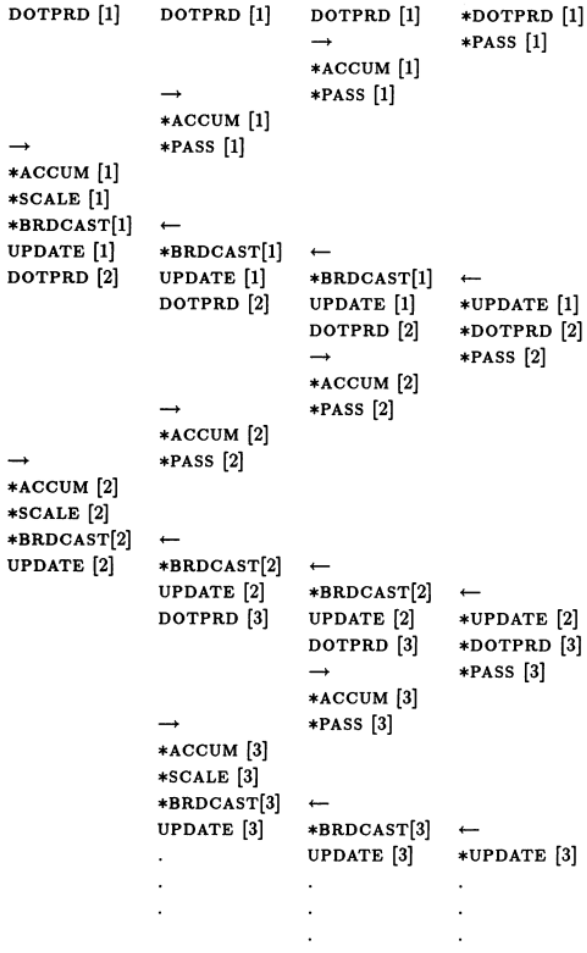
این محاسبات در محیط موازی انجام می‌شود و بهینه‌سازی‌های مختلفی برای کاهش زمان ارتباط بین پردازنده‌ها و بهبود کارایی کلی انجام شده است.

۲.۵ رویکرد موازی‌سازی

کاملا مشابه رویکرد در الگوریتم گرام‌ـاشمیت است

[19] Reflection Vector
[20] Reflection Matrix

۳.۵ توضیح دیاگرام



شکل ۳: نمودار زمان‌بندی برای الگوریتم هاوس‌هولدر موازی

در این دیاگرام، هر ستون نمایانگر فعالیت‌های یک پردازنده در حلقه است و هر سطر نشان‌دهنده گام‌های زمانی متوازی است. فلش‌ها وابستگی‌های بین عملیات در پردازنده‌های مختلف را نشان می‌دهند، که بیانگر نیاز به داده‌های پردازش شده توسط پردازنده‌های دیگر پیش از ادامه فعالیت است.

۴.۵ تحلیل زمانی برای الگوریتم‌ها

زمان اجرای الگوریتم موازی هاوس‌هولدر وابسته به تعداد پردازنده‌ها و ساختار داده‌هایی است که بین آنها تقسیم شده‌است. زمان کلی برای محاسبه R به شرح زیر محاسبه می‌شود:

$$\text{TIME}_R = \sum_{i=1}^{n} (\text{dotprd}[i] + (P_{\text{current}} - 1) \times (\text{ACCUM}[i] + \text{PASS}[i] + \text{BRDCAST}[i] + \text{UPDATE}[i]))$$

که در آن P_{current} تعداد پردازنده‌های فعال در هر گام است. این زمان شامل محاسبه ضرب‌های داخلی، جمع‌آوری نتایج، ارسال داده‌ها بین پردازنده‌ها، پخش نتایج مقیاس‌بندی شده و به‌روزرسانی ماتریس بر اساس نتایج جدید است.

۱.۴.۵ فرمول‌های زمانی مختلف برای تنظیم پردازنده‌ها

بر اساس الگوریتم‌های مختلف و توزیع داده‌ها بین پردازنده‌ها، زمان مورد نیاز می‌تواند متفاوت باشد. برای مثال، در الگوریتم هاوس‌هولدر با توزیع بلوکی:

$$\text{TIME}_{\text{block Householder}} = \frac{n^2p}{2} + \frac{2mn^2}{p} + \frac{np}{2} + \frac{mn}{p} + \frac{3n^2}{4} + \frac{7n}{4} + \frac{n^3p}{4m} - \frac{n^2p}{4m} + n \times \sqrt{n}$$

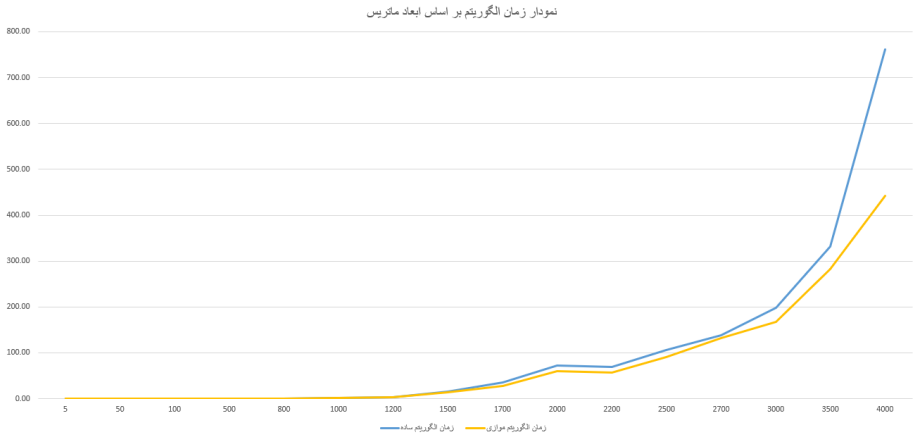
که در آن:

- $\frac{n^2p}{2}$ نشان دهنده زمان پردازش مربوط به مراحل اصلی تجزیه هاوس‌هولدر است.
- $\frac{2mn^2}{p}$ بیانگر زمان لازم برای عملیات میان پردازنده‌ها با توجه به تعداد داده‌های موجود است.
- $\frac{np}{2}$ و $\frac{mn}{p}$ مربوط به توزیع و جمع‌آوری داده‌ها بین پردازنده‌ها است.
- $\frac{3n^2}{4} + \frac{7n}{4}$ زمان های جانبی مربوط به محاسبات غیر متمرکز و پیکربندی‌ها را نشان می‌دهد.
- $\frac{n^3p}{4m} - \frac{n^2p}{4m}$ اصلاحات مربوط به بهینه‌سازی‌های الگوریتمی و حافظه‌ای است.
- $n \times \sqrt{n}$ زمان اختصاصی برای عملیات مرتبط با مقیاس‌بندی و نرمال‌سازی.

$$P_{\text{opt}} = \sqrt{\frac{2m(2n+1)}{V\left((1+4r)n + 6r + 8(r+1)\right)}}$$

که در آن P_{opt} نشان دهنده بهینه‌ترین تعداد پردازنده‌ها برای کمینه کردن زمان اجرا است، با در نظر گرفتن عواملی نظیر توزیع داده‌ها و بار محاسباتی.

۵.۵ نمودار مقایسه الگوریتم ساده و الگوریتم موازی



شکل ۴: نمودار مقایسه زمان اجرای الگوریتم هاوس‌هولدر سریال و موازی بر اساس اندازه ماتریس.

در این نمودار، زمان پردازش الگوریتم هاوس‌هولدر به صورت سری و موازی برای ماتریس‌های مختلف نمایش داده شده است. همانطور که مشاهده می‌شود، با افزایش اندازه ماتریس، الگوریتم موازی بهتر می‌شود.[۲۷]

۶.۵ پیاده‌سازی به زبان سی پلاس پلاس

کد پیاده‌سازی شده برای الگوریتم هاوس‌هولدر موازی، از کتابخانه MPI برای توزیع محاسبات و داده‌ها بین پردازنده‌ها در یک محیط موازی استفاده می‌کند. این روش امکان بهره‌برداری از توان پردازشی چندین واحد پردازشی را فراهم می‌آورد و در نتیجه کاهش قابل توجهی در زمان اجرای کلی الگوریتم دارد.[۲۷]

```cpp
void householderStep(std::vector<std::vector<double>>& R, std::vector<std::vector<double>>& Q, int k, int rank, int size) {
    int m = R.size();
    std::vector<double> x(m - k, 0.0);

    if (rank == k % size) {
        for (int i = k; i < m; i++) {
            x[i - k] = R[i][k];
        }
    }

    // Broadcasting x vector to all processes
    MPI_Bcast(x.data(), m - k, MPI_DOUBLE, k % size, MPI_COMM_WORLD);

    double x_norm = std::sqrt(std::accumulate(x.begin(), x.end(), 0.0, [](double acc, double xi) { return acc + xi * xi; }));
    if (x_norm == 0) return;  // Skip if norm is zero

    std::vector<double> u(m - k);
    if (rank == k % size) {
        x[0] = x[0] > 0 ? x[0] + x_norm : x[0] - x_norm;
        for (int i = 0; i < m - k; i++) {
            u[i] = x[i] / std::sqrt(std::accumulate(x.begin(), x.end(), 0.0, [](double acc, double xi) { return acc + xi * xi; }));
        }
    }

    // Broadcasting u vector to all processes
    MPI_Bcast(u.data(), m - k, MPI_DOUBLE, k % size, MPI_COMM_WORLD);

    // Apply transformation to R and Q matrices
    for (int i = 0; i < m; i++) {
        if (i % size != rank) continue;  // Each process only updates its part of R and Q

        double dot_product = 0.0;
        for (int j = k; j < m; j++) {
            dot_product += u[j - k] * R[i][j];
        }
        for (int j = k; j < m; j++) {
            R[i][j] -= 2 * dot_product * u[j - k];
        }
    }

    // Synchronize before updating Q
    MPI_Barrier(MPI_COMM_WORLD);

    for (int i = 0; i < m; i++) {
        if (i % size != rank) continue;

        double dot_product = 0.0;
        for (int j = k; j < m; j++) {
            dot_product += u[j - k] * Q[j][i];
        }
        for (int j = k; j < m; j++) {
            Q[j][i] -= 2 * dot_product * u[j - k];
        }
    }
}

void householderQR(std::vector<std::vector<double>>& A, std::vector<std::vector<double>>& Q, std::vector<std::vector<double>>& R, int rank, int size) {
    int m = A.size(), n = A[0].size();
    R = A;
    initializeQ(Q, m);

    for (int k = 0; k < std::min(m, n); ++k) {
        householderStep(R, Q, k, rank, size);
        MPI_Barrier(MPI_COMM_WORLD);  // Ensure all ranks finish updating before the next iteration
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::vector<std::vector<double>> A, Q, R;
    int rows, cols;

    // Broadcast the matrix dimensions to all processors
    MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Calculate local rows for each processor
    int localRows = rows / size + (rank < rows % size ? 1 : 0);

    std::vector<std::vector<double>> localA(localRows, std::vector<double>(cols));
    std::vector<std::vector<double>> localQ(localRows, std::vector<double>(cols));
    std::vector<std::vector<double>> localR(localRows, std::vector<double>(cols));

    // Flatten the matrices for MPI operations
    std::vector<double> flatA, flatLocalA(localRows * cols);
    if (rank == 0) {
        flatA.resize(rows * cols);
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                flatA[i * cols + j] = A[i][j];
            }
        }
    }
```

```
    // Scatter the matrix A to all processors
    MPI_Scatter(flatA.data(), localRows * cols, MPI_DOUBLE, flatLocalA.data(), localRows * cols, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Convert flatLocalA to localA
    for (int i = 0; i < localRows; ++i) {
        for (int j = 0; j < cols; ++j) {
            localA[i][j] = flatLocalA[i * cols + j];
        }
    }

    // Perform the Householder QR decomposition
    householderQR(localA, localQ, localR, rank, size);

    // Gather Q and R matrices
    MPI_Gather(localQ.data()->data(), localRows * cols, MPI_DOUBLE, flatQ.data(), localRows * cols, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Gather(localR.data()->data(), localRows * cols, MPI_DOUBLE, flatR.data(), localRows * cols, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

۶ نتایج تجربی

الگوریتم‌ها بر روی چندین ماشین آزمایش شدند که شامل ماشین بی‌بی‌ان باترفلای[۲۱] با ۱۲۸ گره و نقطه شناور نرم‌افزاری[۲۲]، بی‌بی‌ان باترفلای با ۱۶ گره، و همچنین ام‌سی‌ماب[۲۳] با ۱۶ گره. توضیحات معماری هر ماشین به شرح زیر است:

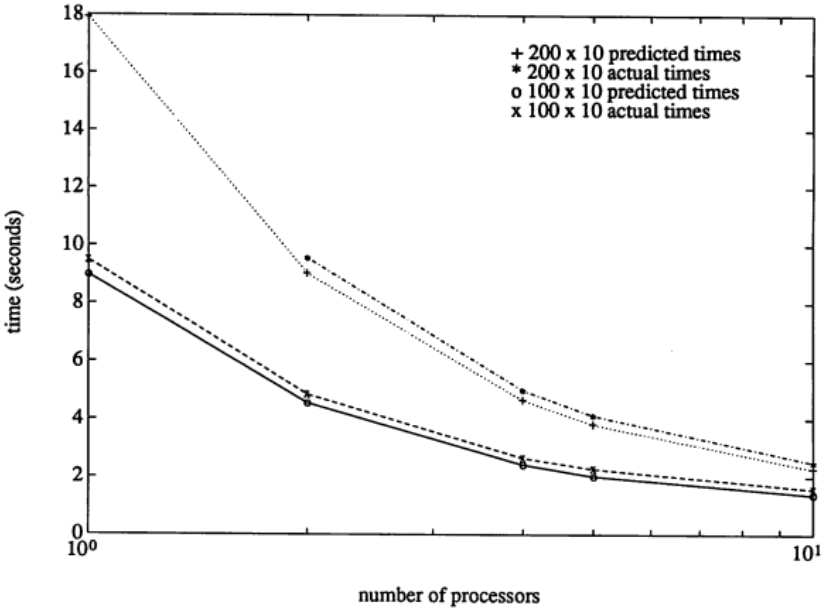
۱.۶ معماری ماشین‌ها

باترفلای: یک ماشین با حافظه مشترک است که هر پردازنده دسترسی مستقیم به ماژول حافظه خود دارد و یک سوییچ سطح $\log_2(p)$ بین پردازنده‌ها و ماژول‌های حافظه برای دسترسی به سایر ماژول‌ها وجود دارد.
ام‌سی‌ماب: یک ماشین طراحی شده و ساخته شده در دانشگاه مریلند است که شامل پردازنده‌های ام‌سی‌ـ ۶۸۰۰۰[۲۴] متصل به یک حلقه شکاف‌دار[۲۵] است، امکان ارتباط مستقیم بین هر زوج پردازنده بدون دخالت یا مداخله از پردازنده‌های دیگر را می‌دهد.

۲.۶ تحلیل نتایج

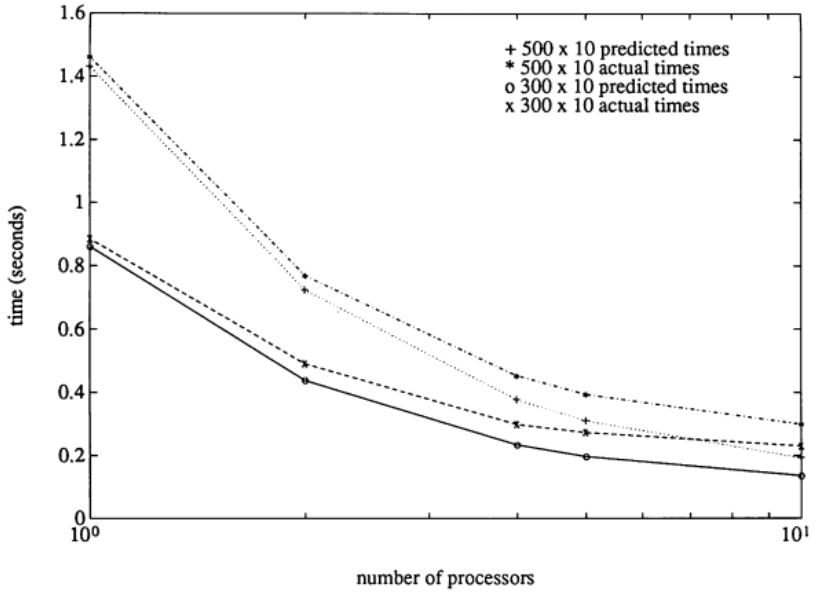
نتایج زمان‌بندی برای الگوریتم‌های گرام‌ـ اشمیت اصلاح شده و هاوس‌هولدر بر اساس مدل‌ها و اندازه‌گیری‌های واقعی انجام شد. تفاوت‌های ناچیزی در داده‌های ام‌سی‌ماب مشاهده شد، اما تغییرات در زمان‌های باترفلای تا ٪۱۰ در دو اجرا با همان پیکربندی پردازنده‌ها و ٪۲۰ با انتخاب متفاوت پردازنده‌ها متغیر بود. نوسانات مشاهده شده به دلیل قطعی‌های سیستمی[۲۶] اتفاق افتاد.
این نتایج بر اساس میانگین ۵ تا ۱۰ اجرا و مقایسه عملکرد الگوریتم‌ها تجزیه و تحلیل شده‌اند.[۱]

[21] bbn Butterfly
[22] software floating point
[23] McMob
[24] MC-68000
[25] slotted ring
[26] system-level interrupts



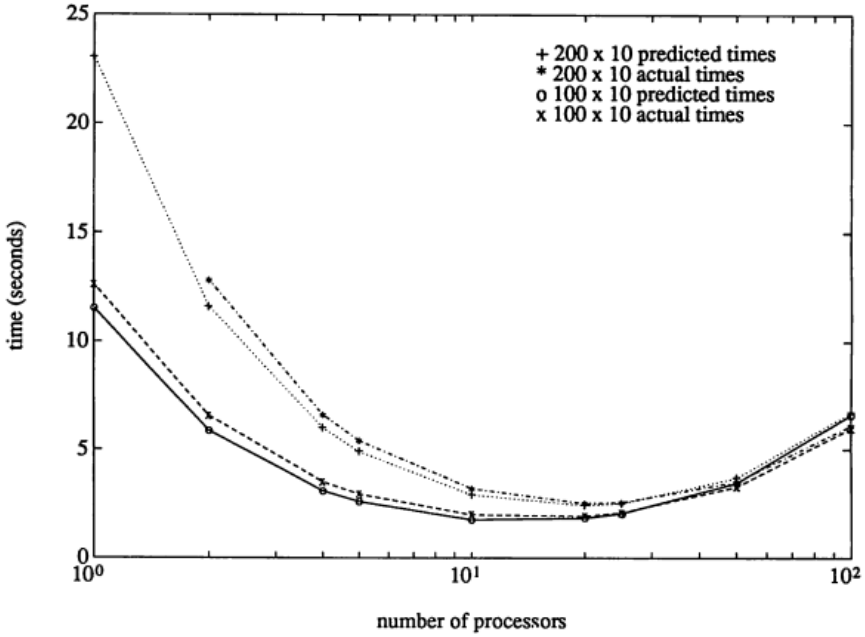
Results of modified Gram-Schmidt timings on 16-node Butterfly, software floating point.

شکل ۵: نتیجه گرام‌ـاشمیت اصلاح شده بر روی بی‌بی‌ان باترفلای با ۱۶ گره



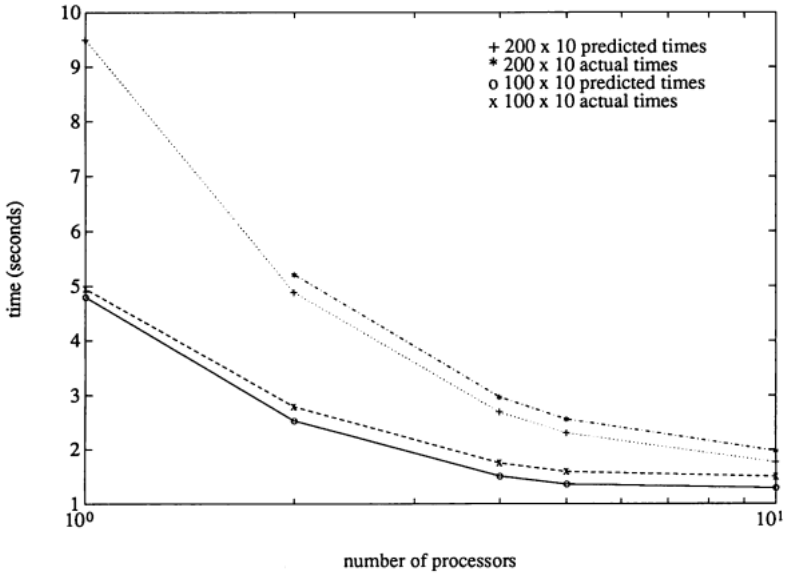
Results of modified Gram-Schmidt timings on 16-node Butterfly, hardware floating point.

شکل ۶: نتیجه گرام‌ـاشمیت اصلاح شده بر روی بی‌بی‌ان باترفلای با ۱۲۸ گره



Results of modified Gram-Schmidt timings on 128-node Butterfly.

شکل ۷: نتیجه گرام‌ـاشمیت اصلاح شده بر روی بی‌بی‌ان باترفلای با ۱۲۸ گره

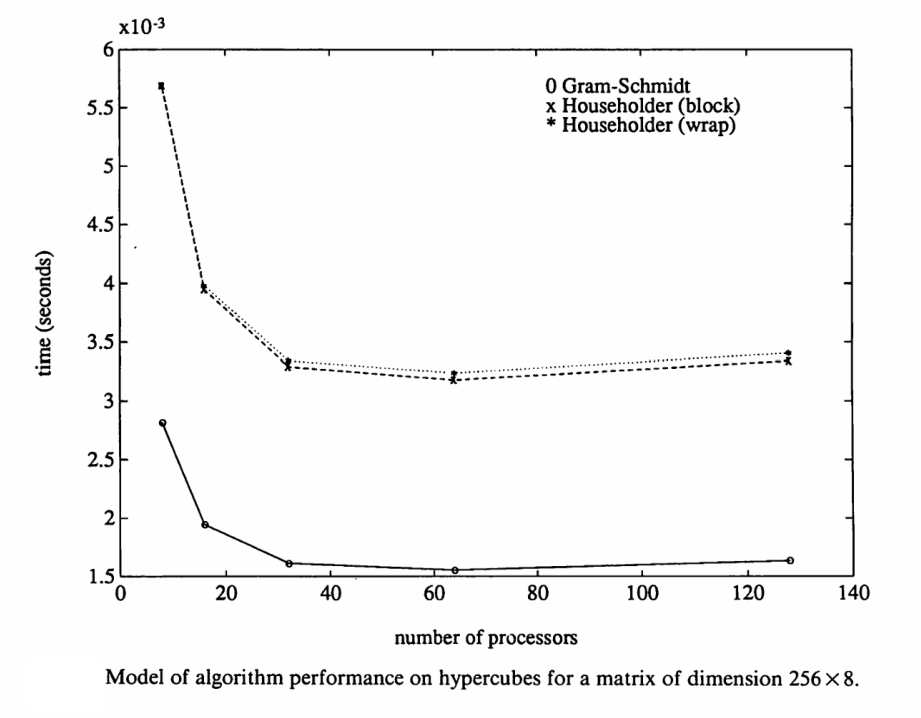


Results of modified Gram-Schmidt timings on 16-node McMob.

شکل ۸: نتیجه گرام‌ـاشمیت اصلاح شده بر روی ام‌سی‌ماب با ۱۶ گره

۷ جمع‌بندی

در این پروژه، به بررسی و مقایسه اجرای عادی و موازی الگوریتم‌های هاوس‌هولدر و گرام‌ـ‌اشمیت بر روی سیستم‌های ارسال پیام برای تجزیه QR پرداختیم و مدل‌های زمان اجرای الگوریتم‌های گرام‌ـ‌اشمیت اصلاح شده و هاوسهولدر برای محاسبه تجزیه QR از ماتریس‌های مستطیلی ارائه و مورد تایید قرار گرفتند.
در تجزیه QR الگوریتم گرام‌ـ‌اشمیت اصلاح شده می‌تواند به مراتب کارآمدتر باشد و در هر حالتی کندتر نخواهد بود. انتخاب الگوریتم برای کاربرد مشخص نیز باید بر اساس خواص پایداری دو الگوریتم و کاربرد مورد نظر انجام شود.



شکل ۹: مقایسه الگوریتم گرام‌ـ‌اشمیت و هاوس هولدر

مراجع

[1] Dianne P. O'Leary and Peter Whiteman, Parallel QR factorization by Householder and modified Gram-Schmidt algorithms

[2] J.L. Barlow and I.C.F. Ipsen, Parallel scaled Givens rotations for the solution of linear least squares problems, SIAM J. Sci. Stat. Comput. 8 (1987) 716-733.

[3] A. Bjorck, Solving linear least squares problems by Gram-Schmidt orthogonalization, BIT 1 (1967) 1-21.

[4] A. Bojanczyk, R.P. Brent and H.T. Kung, Numerically stable solution of dense systems of linear equations using mesh-connected processors, SIAM J. Sci. Stat. Comput. 5 (1984) 95-104.

[5] R.M. Chamberlain and M.J.D. Powell, QR factorization for linear least squares problems on the hypercube, Technical Report CCC 86/10, Chr. Michelsen Institute, Bergen, Norway, 1986.

[6] E. Chu and A. George, QR factorization of a dense matrix on a hypercube multiprocessor, Parallel Comput. 11 (1989) 55-71.

[7] M. Cosnard, M. Daoudi, J.M. Muller and Y. Robert, On parallel and systolic Givens factorizations of dense matrices, in: M. Cosnard et al., eds., Parallel Algorithms and Architectures, (Elsevier Science Publishers B.V., North-Holland, 1986) 245-258.

[8] M. Cosnard, J. Muller and Y. Robert, Parallel QR decomposition of a rectangular matrix, Numer. Math. 48 (1986) 239-249.

[9] J.J. Dongarra, A.H. Sameh and D.C Sorensen, Implementation of some concurrent algorithms for matrix factorization, Parallel Comput. 3 (1986) 25-34.

[10] Lars Elden, A parallel QR decomposition algorithm, Technical Report LiTH-MAT-R-1988-02, Linkoping University, Sweden, 1988.

[11] G.H. Golub, Numerical methods for solving linear least squares problems, Numer. Math. 9: 139-148, 1966.

[12] D.E. Heller and I.C.F. Ipsen, Systolic networks for orthogonal decompositions, SIAM J. Sci. Stat. Comput. 4 (1983) 261-269.

[13] I.C.F. Ipsen, A parallel QR method using fast Given's rotations, Technical Report YALEU/DCS/RR-299, Computer Science Dept., Yale Univ., 1984.

[14] C.R. Katholi and B.W. Suter, A parallel algorithm for computing the QR factorization of a rectangular matrix, Technical Report TR-88-07, Dept. of Computer and Information Sciences, University of Alabama at Birmingham, 1988.

[15] R.E. Lord, J.S. Kowalik and S.P. Kumar, Solving linear algebraic equations on an MIMD computer, J. Assoc. Comput. Mach. 30 (1983) 103-117.

[16] F.T. Luk, A rotation method for computing the QR-decomposition, SIAM J. Sci. Stat. Comput. 1 (1986) 452-459.

[17] J.J. Modi and M.R.B. Clarke, An alternate Givens ordering, Numer. Math. 43 (1984) 83-90.

[18] D.P. O'Leary, Fine and medium grained parallel algorithms for matrix QR factorization, in: H.J.J. te Riele, Th.J. Dekker, and H.A. van der Vorst, eds., Algorithms and Applications on Vector and Parallel Computers (Elsevier Science Publishers B.V., North-Holland, 1987) 347-349.

[19] D.P. O'Leary and G.W. Stewart, Assignment and scheduling in parallel matrix factorization, Linear Algebra Appl. 11 (1986) 275-299.

[20] D.P. O'Leary and G.W. Stewart, Dataflow algorithms for parallel matrix computation, Comm. ACM 28 (1985) 840-853.

[21] D.P. O'Leary and G.W. Stewart, From determinacy to systolic arrays, IEEE Trans. Comput. C-36 (1987) 1355-1359.

[22] D.P. O'Leary, G.W. Stewart and R. van de Geijn, DOMINO: a message passing environment for parallel computations, Technical Report TR-1648, Computer Science Dept., University of Maryland, 1986.

[23] A. Pothen and P. Raghavan, Distributed orthogonal factorizations: Givens and Householder algorithms, SIAM J. Sci. Stat. Comput. 10 (1989) 1113-1134.

[24] A. Pothen, J. Somesh and U. Vemulapati, Orthogonal factorization on a distributed memory multiprocessor, in: M.T. Heath, ed., Proc. Hypercube Multiprocessors 1987, SIAM, Philadelphia (1987) 587-596.

[25] Chuck Rieger, Zmob: hardware from a user's viewpoint, in: Proc. IEEE Computer Society Conf. Pattern Recognition and Image Processing (1981) 484-521.

[26] A.H. Sameh and D.J. Kuck, On stable parallel linear system solvers, J. Assoc. Comput. Mach. 25 (1978) 81-91.

[27] https://github.com/Ali-Noghabi/parallel-QR-Factorization